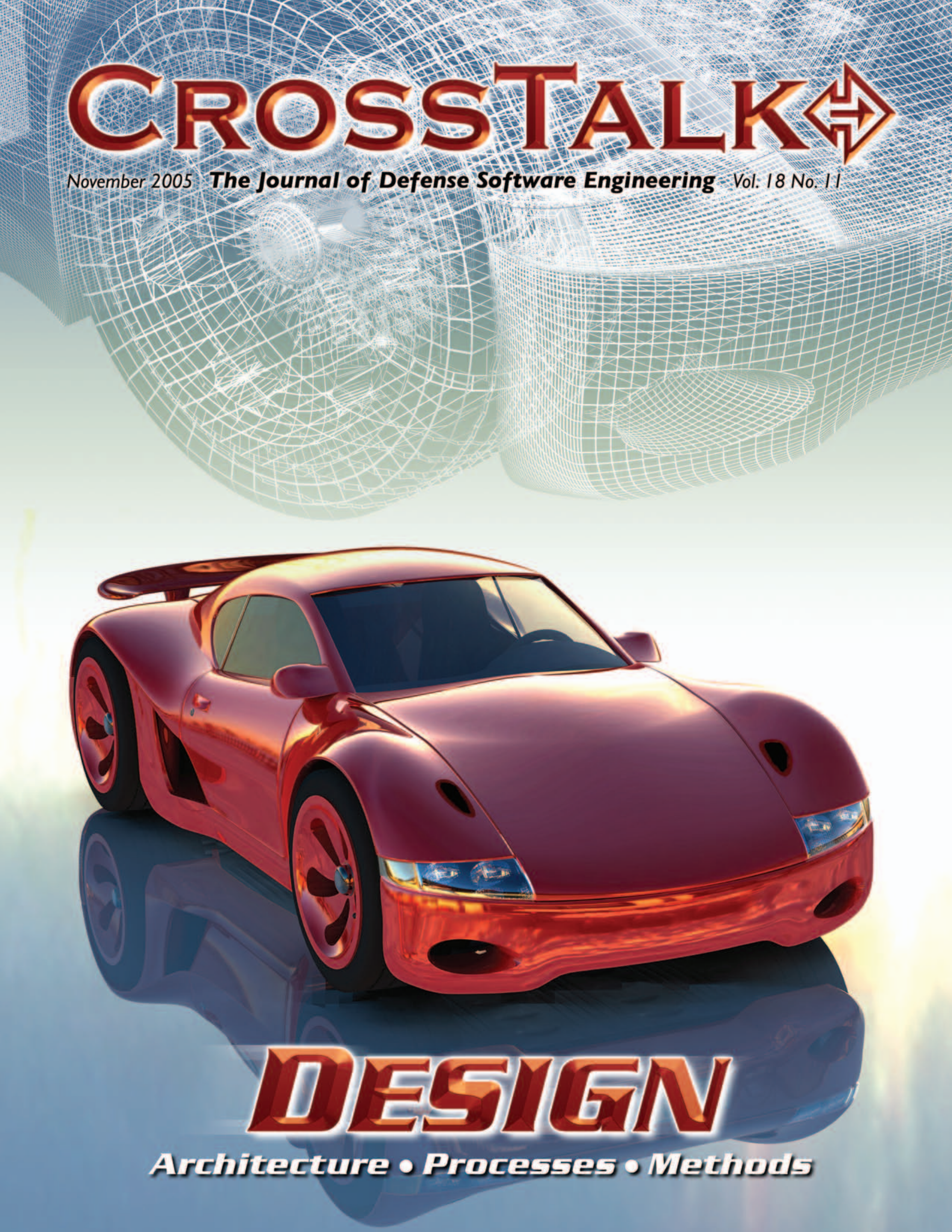
CrossTalk

November 2005 **The Journal of Defense Software Engineering** Vol. 18 No. 11



Design

Architecture • Processes • Methods

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **NOV 2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE **CrossTalk: The Journal of Defense Software Engineering. Volume 18, Number 11, November 2005**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

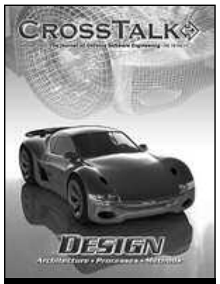
13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18



ON THE COVER
Cover Design by Kent Bingham.

Additional art services provided by Janna Jensen.
jensendesigns@aol.com

CROSSTALK

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Bob Zwitch*
DHS Co-Sponsor	*Joe Jarzombek*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF) and the U.S. Department of Homeland Security (DHS). USAF co-sponsors are the Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor is the National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send subscription correspondence and changes of address to the following address or e-mail us, or use the form on p. 22.

309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Endorsements and Trademarks: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk *Online Services:* See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857, or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

From the Sponsor

Software: Where We've Been And Where We're Going

Has software design finally come of age? Mechanical, electrical, chemical, industrial, and other engineering design disciplines have been in place for centuries and have experienced the associated growth and cross-pollination of shared concepts, tools, trial and error, etc. However, software engineering is still relatively new.

The discipline of software design has only been matured for a few decades. It wasn't until the 1960s that the first software products hit the marketplace. Standards such as American Standard Code for Information Interchange and concepts such as object-oriented code didn't appear until the 1960s. Our dominant programming language C++ didn't emerge until the 1980s. More recently, we have seen design tools such as CASE [Computer-Aided Software Engineering] and fourth generation programming languages.

Our discipline of software engineering has really experienced phenomenal growth right before our eyes. A sign that software design has really arrived is indicated by the hardware changes that are being driven by the complexity of software designs. Our phenomenal growth is also accompanied by real challenges. The increasing complexity of our software also necessitates that our industry stay focused on process improvement. As you read some of this month's articles, take yourself back 10 or 20 years, and revel at how far we have come, but also envision where we need to go.

Kevin Stamey

Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor

From the Publisher

Design Focuses on the *How*

This month we cover another critical phase in the software system life cycle – design. Industry has been successful over time with emphasizing and providing many methods and tools for improving design. Software engineers know their hands will be slapped if they rush from requirements to programming code without taking the time to design. The design phase can be looked at as a problem-solving process. Another way to think of design is to focus on the *how*. Requirements are focused on *what* problem will be solved. The design focuses on *how* the problem will be solved.

Our theme section begins with *Selecting Architecture Products for a Systems Development Program* by Michael S. Russell. This author describes a repeatable process that emphasizes architecture as the source of information that decision makers can turn to throughout the systems engineering process. In *Dependency Models to Manage Software Architecture*, Neeraj Sangal and Frank Waldman relate a new approach using inter-module dependencies to specify and manage system architecture. Next, in *UML Design and Auto-Generated Code: Issues and Practical Solutions* by Ilya Lipkin and Dr. A. Kris Huber, lessons learned from designing with the Unified Modeling Language (UML) are presented from a real-time control system perspective. Lastly, Dr. Hans-Peter Hoffmann describes how systems engineers can capture requirements and specify architecture in *UML 2.0-Based Systems Engineering Using a Model-Driven Development Approach*.

In this month's supporting articles, Arlene F. Minkiewicz and Jeffrey Voas each offer additional information on software security, another important factor to consider during the design phase. I hope this month's issue provides helpful information as new and improved methods continue to evolve design processes.

Tracy L. Stauder

Tracy L. Stauder
Publisher

Selecting Architecture Products for a Systems Development Program

Michael S. Russell
Anteon Corporation

Determining what architecture products are needed to support software development within a larger systems engineering process is a challenge. Existing documents such as the Department of Defense Architecture Framework provide some guidance, but no defined product identification process. The method proposed in this article provides a repeatable process for selecting the architecture products required to support a larger systems engineering effort, defines the content of each product, and identifies each product's customer.

The purpose of architecture is to answer questions. In other words, the architecture provides the information needed by decision makers during the course of the systems engineering process to define concepts and processes, allocate functionality, define test metrics, design software, and make other development decisions. However, the output of the systems engineering (SE) process is not the architecture; rather, it is the system being produced [1]. Unfortunately, the architecture often becomes divested from the SE process it is meant to support, becoming an output or deliverable in and of itself [2].

In part, this is perpetuated by government regulations specifying that each acquisition program produce architecture with little thought as to how architecture will be used by the program [3, 4, 5]. So, the output of most architecture efforts tends to be a three-ring binder that weighs five pounds or so, which no one ever reads. This has given a bad name to the architecting process, and has left many decision makers asking why they spent their limited money and time producing architectures.

To correct this situation, both the software-centric and system-centric communities need to reexamine the architecting process, rediscover the intended uses for architecture, and ensure architecting is always done in support of the SE life cycle. For software designers, this will mean a break from the concept that four or five standard Unified Modeling Language (UML) diagrams will solve the needs of all stakeholders. This article presents one method for ensuring the architecture is producing the products needed to support the overall SE process.

Tying Architectures Into the SE Process

The SE process has been described as an elaborate engineering decision process that includes the following [6]:

- Begins with understanding the system requirements and specifications.
- Translates these specifications into a conceptual design in the form of a functional architecture.
- Translates this functional architecture into logical design or physical architecture.
- Translates the physical architecture into a detailed design or implementation architecture for the system ultimately to be produced or acquired.
- Completes development through the production of a product that conforms to this architecture, potentially through various strategic sourcing efforts and associated integration and test.

"Developing architecture as part of a program's SE [systems engineering] process can be a critical component of the program's success, but only if the right products are identified and produced at the right time for the right customer."

These steps show the SE process for what it really is – a series of decision points leading up to a delivered product. This means the architecture's products must be tied to the rest of the SE process and, more specifically, to decision points within that process. This allows the architect to determine what products need to be produced, when to produce them, and what level of detail needs to be present. To tie architecture products to decision points, the architect must understand what decisions need to be made. It is a process that begins with these questions:

1. What is the output of my SE process?
2. What decisions need to be made to produce the output?
3. What sort of information is needed to make those decisions?
4. Which architecture products provide the information in a format that is understandable to the decision maker?
5. When are the architecture products needed?
6. How should my project be staffed to produce those products?
7. Which software tools will help the architect build those products?

For example, if the output of the SE process is an electronic timecard system, then decisions include these: "What are the expectations of its users?" "How many employees will use it concurrently?" "What technology options exist?" and "Which technical option works best with the existing timekeeping business process?" Concerning the time card software itself, the decision maker would probably like to know the advantages and disadvantages of different ways of entering time card data, ease of updating and maintenance, and how the software will interoperate with the business' existing software.

Each of these decisions is made at a different point in the SE process, and many of the decisions are interdependent. It is the architect's job to recognize this and suggest the appropriate mix of architecture products that will provide the needed information – in essence, answer the questions. Figure 1 describes this process.

Please note that the last step in the process concerns what software tools are needed to build the architecture products. Too often, a project starts with the notion, "We're using XX company's tool," without considering how that software will be used or even if it can be used within the program's existing SE process. In fact, many of these tools assume an SE process, and the software is optimized to produce architecture products designed for that process (i.e., the Rational Unified Process) [7]. While this is not necessarily a bad thing, it does mean that the chosen tool is driving product selection and the overall process to produce the architecture products. Sometimes this situation is directed such as a program that is required to use a model-driven architecture [8] approach. Ideally, the software tool should be chosen after the products to be built have been identified.

Remember that decisions made early in the SE process require different information than those decisions made closer to the end. For instance, describing the business process that a new system will support is done early on, while defining software to support that process is done later. Understanding when information is useful to a decision maker is just as important as understanding what information he or she needs.

Finally, a system has many customers. The electronic timecard system described in the earlier example has an obvious customer: the employee. However, there are many other customers such as the system developers, the company's executives, time card system vendors, and the company's accounting department. Each of these customers has an interest in the new system, but each makes different decisions and needs different information to make those decisions. A well-built architecture takes all of these needs into consideration and is capable of producing the right architecture product, in the optimal format, to meet each customer's need.

The Product Identification Process

The product identification process outlined in this article is a matrix-based approach used successfully by several DoD organizations over the past five years, and is also taught in the Federal Enterprise Architecture Certification (FEAC) Institute's [9] certification program. The matrices can be produced with most any office automation or database software. However, the matrix is normally put together in a spreadsheet. To help examine how the process would be implemented, a completed set of matrices using the electronic timecard example are included in the online version of this article at <www.stsc.hill.af.mil/crosstalk/2005/11/0511Russell.html>. The matrices presented below contain a subset of the same information.

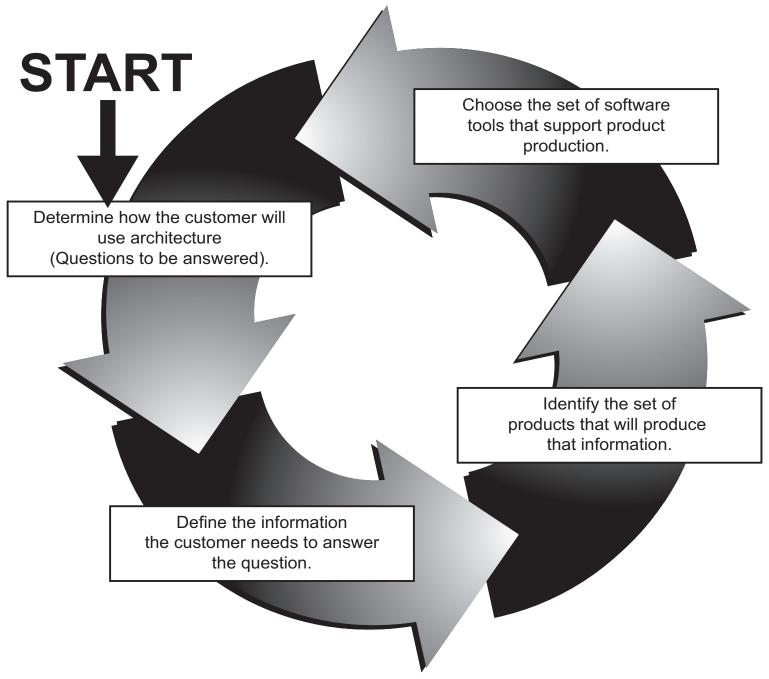

START
Determine how the customer will use architecture (Questions to be answered).
Choose the set of software tools that support product production.
Identify the set of products that will produce that information.
Define the information the customer needs to answer the question.

Figure 1: *Product Identification Process*

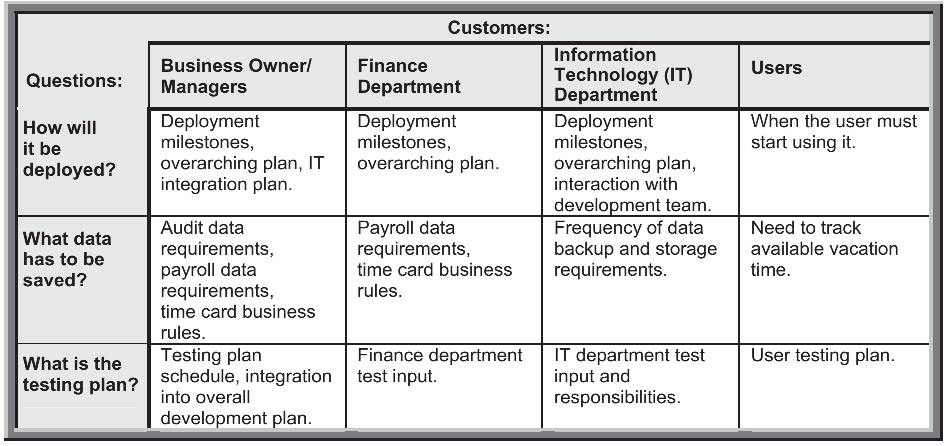
Step One

Step one focuses on identifying the architecture's customers and determining each customer's questions or decisions they need to make. In the Customer-Question Matrix (Figure 2), the customers are listed along the top axis and a list of questions along the left axis. Next, the information needed to answer each customer's question is added to the matrix. Developing this matrix can be greatly improved by letting each customer see the responses of other customers. This typically results in less question redundancy, especially when the information needs between customers are similar.

Step Two

In step two, the focus shifts from the customer's questions and decision points to which architecture products provide the information needed to answer the questions or support a decision. It is important to remember that even when several customers need similar information, the architecture product and the level of detail for that product might be different. For example, a business executive and a programmer would both need to understand the concept for the company's new timecard

Figure 2: *Customer-Question Matrix*



	Customers:			
Questions:	**Business Owner/ Managers**	**Finance Department**	**Information Technology (IT) Department**	**Users**
How will it be deployed?	Deployment milestones, overarching plan, IT integration plan.	Deployment milestones, overarching plan.	Deployment milestones, overarching plan, interaction with development team.	When the user must start using it.
What data has to be saved?	Audit data requirements, payroll data requirements, time card business rules.	Payroll data requirements, time card business rules.	Frequency of data backup and storage requirements.	Need to track available vacation time.
What is the testing plan?	Testing plan schedule, integration into overall development plan.	Finance department test input.	IT department test input and responsibilities.	User testing plan.

Information Needed by Each Customer:	Architecture Products:					
	System Concept and Program Plan	Activity and Process Diagrams	Business Rules Description	Web-Portal Screen Mock-up	Regulation Compliance Traceability Matrix	Data Model
System Concept Information	White paper			Graphics		
Federal and State Timekeeping Regulation Compliance			Text document		Spreadsheet	
Data Needed for Auditing, Payroll, Vacation Tracking, and Other Requirements		UML activity diagram	Data flow diagram	Graphics	Spreadsheet	UML class diagram
Time Card Usage, Review, and Auditing Business Rules		UML activity diagram	State chart		Spreadsheet	UML class diagram

Figure 3: *Product-Information Matrix*

system, but the level of detail needed by each would be much different.

For this Product-Information Matrix (Figure 3), the information needed by each customer is listed on the left axis, and architecture products that could provide that information are listed along the top axis. Quite often, the information needed will span several architecture products. For instance, determining who can update a company's human resources database and under what conditions an update can occur may require activity diagrams, a logical data model, a business rules model, and other architecture products.

When possible, information needs as described by the customers should be consolidated so the matrix does not become unwieldy, but only when the information needed is at the same level of detail. Although two customers may need the same type of information to answer their question, the information they need may reflect a different level of detail. For example, if a business owner wants to understand database access permissions, then a business rules model might be the best architecture product to provide the information. However, the database developer would need a data model based on the business rules to build the database. Few business owners will look at or understand a data model, but it is the right product for the developer.

Step Three

The next step in the process matches each customer with the set of architecture products they need to answer questions and make decisions in a timely manner. This step gives the architecture developers the information they need to plan the architecture development process and integrate it into the overall SE process. The Customer-Product Matrix (Figure 4) supports this mapping, identifies when each product is needed, and highlights dependencies between products. In this matrix, the customers are again listed along the top axis, while the architecture products are listed on the left axis. Delivery dates, product formats, i.e., activity diagram versus data flow diagram, metrics, and other information should be used to indicate the mapping between customers and products.

It is likely that a specific product such as the *activity model* will be identified as being used by multiple customers. Keep in mind that each customer will probably use his or her activity model in a different manner, which should be captured in the *information* section of the first matrix. This means that there probably will not be one activity model for the architecture. Rather, there will be several views of the activity model that are relevant to a specific customer's information needs, derived from the same pool of architectural data.

Figure 4: *Customer-Product Matrix*

Architecture Products:	Customer:			
	Business Owner/ Managers	Finance Department	IT Department	Users
System Concept	DEC	DEC	DEC	
Activity and Process Diagrams	JAN – High Level APR – Final	JAN – High Level MAR – Detailed	JAN – High Level MAR – Detailed	MAR – Detailed
Business Rules Description	JAN – Initial	MAR – Detailed	JAN – Initial MAR – Detailed APR – Final	
Regulation Compliance Traceability Matrix	APR			
Data Model		MAR – Business Rule Text	FEB – UML Class Diagram	

Step Four

Now that architecture products needed to support the SE process have been identified, the last step is to choose the software tools that the architects will use to produce products in a format useful for each customer's decision-making process. Typically, an executive-level decision maker will not want to look at a product as displayed in a Computer Aided Systems Engineering (CASE) tool, while a software developer will not get the information they need from a Power Point presentation. So the architecture should rely on a suite of tools to produce, store, and display the architecture's products. Each tool serves a defined purpose within the architecture, and together they support the creation and integration of the architecture.

Generally, there are four types of software tools used to support architecture development: CASE tools, databases, executable modeling tools, and Web sites. Normally, one tool of each type is needed because each customer has different needs for viewing and using his or her architecture products. One word of caution, even though many vendors will try to sell a *one-size-fits-all* software solution, very few tools support all aspects of your development process. So choose the tool that provides the information needed by that customer to support his or her decision-making process. Do not expect a customer to modify his or her SE process or decision-making criteria just because your software cannot deliver the architecture in a form he or she can use.

Other Uses for the Matrices

Understanding what products to produce is only half the challenge. The next challenge is determining how to staff the development project, including how many people to hire and what skills those people should have. The information contained in the matrices can also help answer these questions. For example, data models are generally produced later in the SE life cycle than system use cases are produced, meaning a project may not need to hire a data modeler at the very beginning of the project. Likewise, a completed set of matrices will normally show a need for several systems analysts toward the beginning of the life cycle, one or two during the middle, and a large group at the end during the test and evaluation stage.

So, by matching the expected delivery dates for each architecture product captured in the matrix, a good initial staff skill set and loading matrix can be produced. Staffing choices are not always intuitive, and the completed matrices can significantly help the program manager justify his project funding requirements by having hard numbers to base his staffing plan upon.

Conclusion

The purpose of architecture is to provide the information needed by decision makers to make decisions. Developing an architecture as part of a program's SE process can be a critical component of the program's success, but only if the right products are identified and produced at the right time for the right customer.

The matrix-based approach outlined here has proven to be a repeatable and successful process for its users. When used early in the SE life cycle, it helps focus developers on how the architecture should be used by zeroing in on the questions to be answered and the information needed to answer those questions. During production, it helps set customer expectations concerning the types of products to be produced, the level of effort and skills required to produce the products, and when each product will be delivered. At any time during the life cycle, the information can be used to show what decisions need to be made and what architectural information these decisions should be based on.

Lastly, the matrices form a body of knowledge that can be reused when planning future projects.◆

References

1. Rechtin, Eberhardt, and Mark Maier. The Art of Systems Architecting. Boca Raton, FL: CRC Press, 1997.
2. Maier, Mark, David Emery, and Rich Hilliard. "ANSI/IEEE 1471 and Systems Engineering." Systems Engineering 2.3 (1990): 168-176.
3. Office of the Secretary of Defense: Acquisition Technology and Logistics. DoD 5000.1 The Defense Acquisition System, Washington: Department of Defense, 2003.
4. The Joint Staff. CJCSI 3170-01E, The Joint Capabilities, Integration and Development System, Washington: Department of Defense, 2005.
5. Office of the Secretary of Defense National Information Infrastructure. The DoD Architecture Framework Vers. 1, Washington: Department of Defense, 2004.
6. Sage, Andrew, and Charles Lynch. "Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives." Systems Engineering 1.3 (1998): 176-227.
7. Kruchten, Philippe. The Rational Unified Process. Reading, MA: Addison Wesley Longman Inc., 1998.
8. Miller, Joaquin, and Jishnu Mukerji. MDA Guide Vers. 1.0.1. Needham, MA: Object Management Group, 12 June 2003.
9. The Federal Enterprise Architecture Certification Institute. FEAC Institute. 25 Feb. 2005 <www.feacinstitute.org>.

About the Author

Michael S. Russell is the director of Enterprise Architectures for the Anteon Corporation's Systems Engineering Group. He has served as lead architect on numerous federal, Department of Defense, and industry enterprise architecture efforts. He is a lecturer with the Federal Enterprise Architecture Certification Institute and is a member of the International Council on Systems Engineering. He has taught courses in systems engineering for the past seven years, and has published several articles on systems engineering topics. He has a Master of Science in systems engineering from George Mason University.

Anteon Corporation
2231 Crystal DR STE 600
Arlington, VA 22202
Phone: (703) 769-6160
E-mail: mrussell@anteon.com

Coming Events

December 4-7
WSC '05
Winter Simulation Conference
Orlando, FL
www.wintersim.org

December 5-8
The 26th Institute of Electrical and Electronics Engineers Real-Time Systems Symposium
Miami, FL
www.rtss.org/rtss2005.html

December 5-9
Annual Computer Security Applications Conference ACSAC 2005
Tucson, AZ
www.acsac.org

December 7-9
I-SPAN 2005
International Symposium on Parallel Architectures, Algorithms, and Networks
Las Vegas, NV
http://sigact.acm.org/ispan05

December 11-14
ICIS 2005
International Conference on Information Systems
Las Vegas, NV
http://icis2005.unlv.edu/

December 19-21
Collaborate COM 2005
San Jose, CA
www.collaboratecom.org/index.php

May 1-4, 2006
2006 Systems and Software Technology Conference

STC Systems & Software Technology Conference

Salt Lake City, UT
www.stc-online.org

Dependency Models to Manage Software Architecture

Neeraj Sangal and Frank Waldman
Lattix, Inc.

This article describes a new approach for managing software architectures. It uses inter-module dependencies to specify and manage the architecture of software applications. The technique, based on a matrix representation, is simple, intuitive, and appears to scale far better than the directed graph representations that are used currently. It enables specification and automatic enforcement of architectural intent such as layering and componentization. The article concludes by showing how this approach can be applied to a real application. We build a dependency model to represent the architecture of Ant, a popular Java build utility. We then examine how Ant's architecture has evolved over several versions of the software.

Software development has a way of becoming difficult over time. While they often start well, software projects begin to bog down as enhancements are made to meet new demands and as development teams change. It takes longer to fix problems because fixes made in one area end up introducing bugs in other areas. In no time, a project becomes so complex that no single engineer understands the whole system. As a result, original design assumptions are lost and the boundary between various parts of the system begins to blur. Systems that started out as modular become monolithic.

A powerful new approach, based on using the Dependency Structure Matrix (DSM), has recently been proposed for specifying software architectures. This approach has been built upon ideas that have actually been around for a very long time. The basic notion of *divide and conquer* has been around since time immemorial. Engineers take a large system and decompose it into subsystems; when a subsystem itself becomes too large, it is in turn split up in a process called *hierarchical decomposition*. The decomposition is done in such a way that closely coupled subsystems are closer to each other, while loosely coupled subsystems are kept farther apart.

The problem in software systems is that it is very easy for developers to create undesirable couplings between subsystems. Often this is done without an understanding of the overall system. As a result, subsystems become tightly coupled over time.

The new approach has two key elements: (1) a precise hierarchical decomposition and (2) explicit control over allowed and disallowed dependencies between the subsystems. DSM is ideal for this approach because it provides a compact representation that can easily scale up to tens of thousands of classes or files, whereas conventional box-and-arrow diagrams become unusable for systems composed of even a few hundred classes.

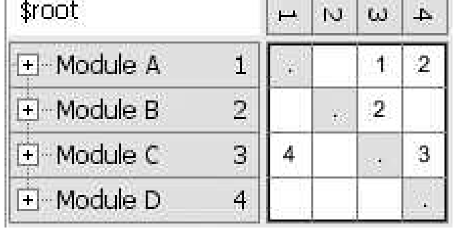
Understanding DSM

Brief History

DSM has traditionally been used to model tasks involved in the development of discrete products. The structure of dependencies between tasks helps in understanding which tasks can be done in parallel and which have to be performed sequentially. Cyclic dependencies are indicators of possible rework that may be necessary. Steven Eppinger at Massachusetts Institute of Technology's Sloan School [1] spearheaded the application of DSM within some of the largest companies in the world, including Boeing, General Motors, Intel, and many others. However, applying DSM to software is new.

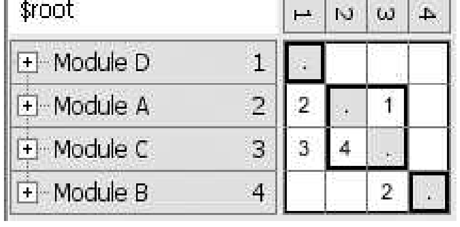
Software engineers have always understood the importance of dependencies between modules. However, they have tended to visualize the modules and their dependencies as directed graphs, i.e., box-and-arrow diagrams. The Unified Modeling Language (UML) makes extensive use of directed graphs, where a variety of boxes and line types are used to indicate the many types of relationships that might exist between different types of modules. This is useful for detailed design but quickly becomes unwieldy as the size of the application increases.

Figure 1 A-B: *A Simple DSM Before and After Partitioning*



$root		1	2	3	4
Module A	1	.		1	2
Module B	2		.	2	
Module C	3	4		.	3
Module D	4				.

Figure 1A



$root		1	2	3	4
Module D	1	.			
Module A	2	2	.	1	
Module C	3	3	4	.	
Module B	4			2	.

Figure 1B

UML diagrams provide limited value in controlling the actual implementation. Indeed, far from controlling the design of the application, developers feel burdened when they have to do a round trip back to their UML models to keep them synchronized with code. Many development teams simply stop maintaining their UML models beyond the initial stages of the project.

Using matrices in systems engineering has a long history. Systems engineers have used N-squared diagrams to model the input and output flows within their system. Subsequent work by Baldwin and Clark at Harvard Business School [2] employing DSM to model the evolution of the computer industry brought it one step closer to the engineering of software.

The key ideas underlying our approach [3], like most ideas in dependency tools, are not new. The notion of inter-module dependency was articulated by Parnas in his early papers (most notably [4]), and the extraction and exploitation of dependencies has been the subject of many more recent projects. The potential significance of the DSM for software was noted by Sullivan et al. [5] in the context of evaluating design tradeoffs. Similarly, Lopes and Bajracharya [6] have also applied DSM to study the value of aspect-oriented modularization. MacCormack et al. [7] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux.

Our approach, however, is the first application of DSM for the explicit management of inter-module dependencies. More information about DSM and available tools, including those from Lattix, can be found at <www.dsmweb.org>.

Making Sense of DSM

Figure 1A shows a simple DSM for a system that consists of four subsystems labeled Modules A, B, C, and D. In the square matrix, the row and column number represent the same module (for compactness, only the rows are labeled). The cells in the grid show the strengths of the interdependencies between each module.

The way to read a DSM is to read the dependencies down a column. For instance, column 1 shows that Module A depends on Module C with dependency strength of 4. Correspondingly, reading across row 1 tells us that Module A provides to Module C and Module D with dependency strengths of 1 and 2 respectively.

Traditionally, DSM has often used an X to indicate a dependency. Also note that a large part of DSM literature reverses our convention of how rows and columns are used. They use rows to indicate *dependencies* and columns to indicate *provides*. However, we have found that our convention is a little more intuitive for software engineers and is also consistent with N-squared diagrams.

Figure 1B shows the DSM after partitioning. Partitioning is a special operation that reorders and regroups modules. The modules are ordered in such a way that those modules that *provide* to other modules are placed at the bottom of the DSM, while modules that *depend* on other modules are placed at the top. If there were no dependency cycles, this would yield a *lower triangular* matrix, i.e., one without any dependencies above the diagonal.

Partitioning also groups together those systems that have dependency cycles. In this case, Modules A and C depend on each other and therefore have been grouped together. This form of the matrix is called *block triangular* because it has been split up into three blocks in which there are no dependencies above the diagonal. Layered systems are naturally expressed as lower triangular matrices.

The grouping of modules can also be shown in different ways. A new compound module can be formed by merging Modules A and C as shown in Figure 2A, after which the matrix becomes *lower triangular*. Notice also that Module D now depends upon the new Module A-C with dependency strength of 5, which is an aggregation of Module D's dependency on both Module A and Module C.

Furthermore, the identities of the basic modules can still be retained by introducing a hierarchy, as in Figure 2B, in which the grouping of A and C is shown by their indentation. The hierarchical decomposition shows that the system has been decomposed into three subsystems: Module D, Module A-C, and Module B. Module A-C is in turn decomposed into Module A and Module C.

This might seem like a simple example but hierarchy is key to scaling with DSM. Hierarchy enables DSM to conveniently model systems with thousands of classes. Hierarchy is also important to the succinct definition of design rules that are used to specify allowed and disallowed dependencies. Design rules can be used to specify architectural patterns such as layering, componentization, external library usage, and other dependency patterns between subsystems. When DSM is combined with design rules, they are called dependency models.

Defining a Dependence Relationship

Before DSM can be applied to software, we need to define the meaning of the statement "a module is dependent upon another module." DSM is a general device that leaves the choice of the definition of dependency to its user. In this case, we are interested in the architecture of the software from *a developer's perspective*. This perspective is important if we are to keep the design modular and to prevent the complexity from spiraling out of control over time.

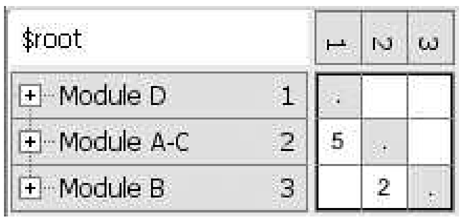
Therefore, we say that Module A depends upon Module B if the developer of Module A needs to know about the behavior of Module B. The good thing about this definition for software engineering is that, except in a few cases, this can generally be deduced by automatic analysis directly from source code of languages such as Ada, C/C++, and Java. The automatic analysis can be accomplished by utilizing commercially available programs to extract the dependencies. For newer languages such as Java and C#, the dependencies can even be extracted from byte code.

For example, in Java we define a Class A as being dependent on Class B if the following occurs:

1. Class A inherits from Class B (implements in the case of an interface).
2. Class A calls a method or a constructor in Class B.
3. Class A refers to a data member in Class B.
4. Class A refers to Class B (e.g., as in an argument in a method).

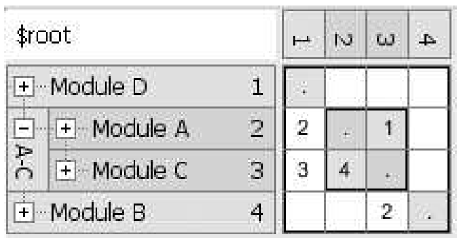
The dependency strength can be calculated in a variety of ways. One is to look at

Figure 2A



$root		1	2	3
Module D	1	.		
Module A-C	2	5	.	
Module B	3		2	.

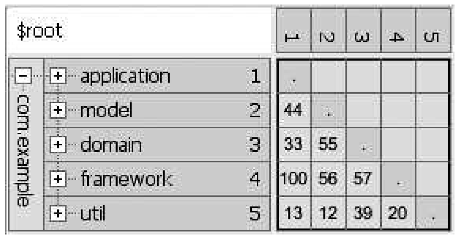
Figure 2B



$root			1	2	3	4
Module D		1	.			
A-C	Module A	2	2	.	1	
	Module C	3	3	4	.	
Module B		4			2	.

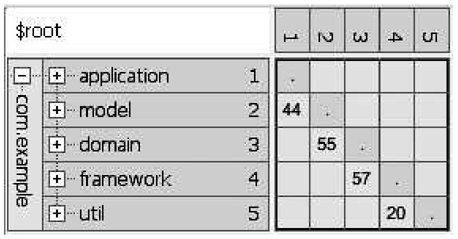
Figure 2 A-B: *The Regrouped DSM and Its Hierarchical Expansion*

Figure 3 A-D: *Architecture Patterns in a DSM*



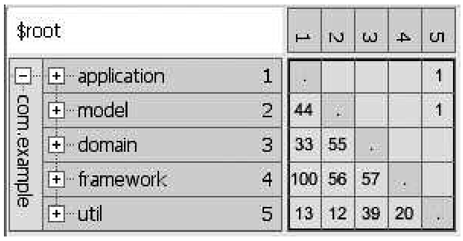
$root			1	2	3	4	5
com.example	application	1	.				
	model	2	44	.			
	domain	3	33	55	.		
	framework	4	100	56	57	.	
	util	5	13	12	39	20	.

3A: Layered Pattern



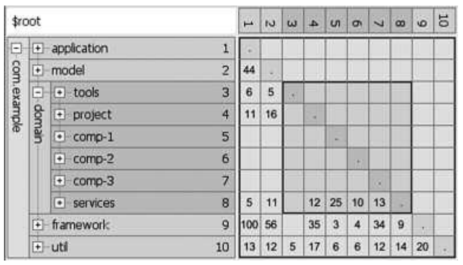
$root			1	2	3	4	5
com.example	application	1	.				
	model	2	44	.			
	domain	3		55	.		
	framework	4			57	.	
	util	5				20	.

3B: Strictly Layered Pattern



$root			1	2	3	4	5
com.example	application	1	.				1
	model	2	44	.			1
	domain	3	33	55	.		
	framework	4	100	56	57	.	
	util	5	13	12	39	20	.

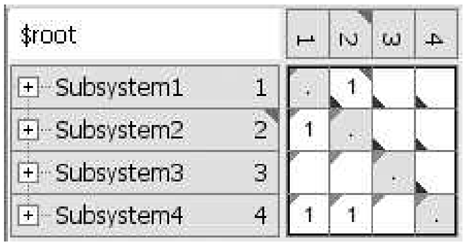
3C: Imperfectly Layered Pattern



$root				1	2	3	4	5	6	7	8	9	10
com.example	application		1	.									
	model		2	44	.								
	domain	tools	3	6	5	.							
		project	4	11	16		.						
		comp-1	5					.					
		comp-2	6						.				
		comp-3	7							.			
		services	8	5	11		12	25	10	13	.		
	framework		9	100	56		35	3	4	34	9	.	
	util		10	13	12	5	17	6	6	12	14	20	.

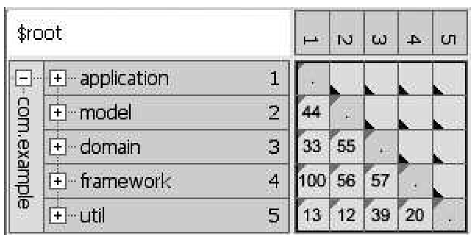
3D: Component Pattern

Figure 4A



$root	1	2	3	4
Subsystem1 1	.	1		
Subsystem2 2	1	.		
Subsystem3 3			.	
Subsystem4 4	1	1		.

Figure 4B



$root		1	2	3	4	5
com.example	application 1	.				
	model 2	44	.			
	domain 3	33	55	.		
	framework 4	100	56	57	.	
	util 5	13	12	39	20	.

Figure 4 A-B: *DSM with Design Rules*

the number of classes each class depends on; that number is then aggregated in the hierarchy. Yet another possible way is to calculate the dependency strength based on the number of actual uses each class makes of other classes; that number is then aggregated in the hierarchy. Furthermore, it is sometimes useful to filter out specific dependencies such as class references because eliminating them is easy.

Architecture Patterns in a Dependency Model

Layering and Componentization in a DSM

The DSM representation is uniquely suited for representing certain architectural patterns. Layering is one such pattern. In fact, even when the layering is imperfectly implemented it can still be recognized in a DSM.

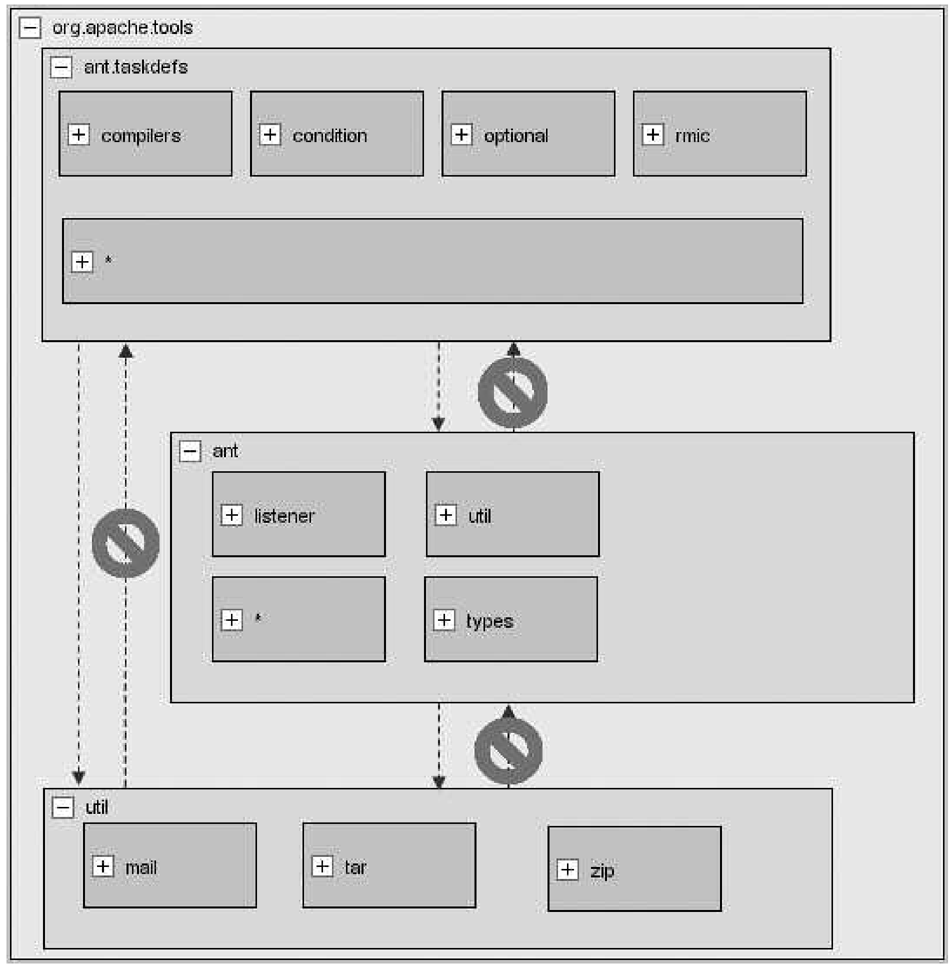
Figure 3A shows the example of a layered system. The figure illustrates that the system consists of five subsystems: *application*, *model*, *domain*, *framework*, and *util*. The DSM shows that the layer at the bottom, util, does not depend on any of the other subsystems; *framework* depends on *util*; *domain* depends on *framework* and *util*; and so on. The lower triangular nature of the matrix makes it immediately apparent that this is a layered system. Figure 3B shows a strictly layered system where each layer depends only on the preceding layer.

Finally, Figure 3C shows an imperfectly layered system. Since the DSM is not lower triangular even after partitioning, we know that there are cyclic dependencies. In this case, the dependencies in column 5 indicate that *util* has dependencies on *application* and *model*. However, the imbalance between the strength of the dependencies suggests that this is an imperfectly layered system.

This discussion must not be construed to suggest that every software application should have a layered design. While a majority of large applications are layered, DSM itself makes no prescription about whether applications should always be layered. However, for those applications that are layered, a representation based on DSM is natural and powerful. As we shall see shortly, design rules allow these architectural patterns to be enforced quite easily.

Figure 3D shows private subsystems *comp-1*, *comp-2*, and *comp-3* within subsystem *domain*. The DSM reveals that nothing in the system depends on these private subsystems. Furthermore, the DSM illustrates that these private subsystems do not depend on each other. This suggests that it is likely that they could be worked upon in parallel once the framework that they depend on is in place.

Figure 5: *Conceptual Architecture of Ant*



Design Rules: Enforcing Architectural Patterns

The architectural patterns that were described in the previous section can be enforced using design rules. The underlying concept of design rules is quite simple: Once you represent the architecture in a hierarchical DSM, then specifying which cells are allowed or disallowed from having dependencies is a natural extension to document and communicate the design intent.

When design intent in the form of design rules is added to a DSM, the result is a dependency model. This dependency model communicates not just what the actual dependencies are but also the allowed and disallowed dependencies. The matrix representation provides a succinct and intuitive visualization for design rules. Figure 4A shows a DSM with design rules expressed as triangles in the corners of the cells. The upper left triangle represents an allowed dependency, while the lower left triangle represents a disallowed dependency and the upper right triangles represent design rule violations. The use of colors can enhance usability; for instance, the triangles can be colored green, yellow, or red to indicate an allowed dependency, a disallowed dependency, or a violation, respectively.

If the DSM grid represents the design space, the design rules qualify that design space by specifying which parts of the design space are allowed to have dependencies and which are disallowed. In a system with 1,000 classes, a fully expanded DSM grid has one million cells. Since each cell represents design intent, there are one million possible design rules. Fortunately, classes interact with each other in fairly regular ways. Layers are just one example of how classes within each layer interact with classes in other layers. For a five-layer system, just five rules are needed to specify their interaction regardless of the number of classes within the system. Figure 4B shows the design rules for enforcing the layers in such a system. Note that showing only the cannot-use rules tends to make the DSM more readable.

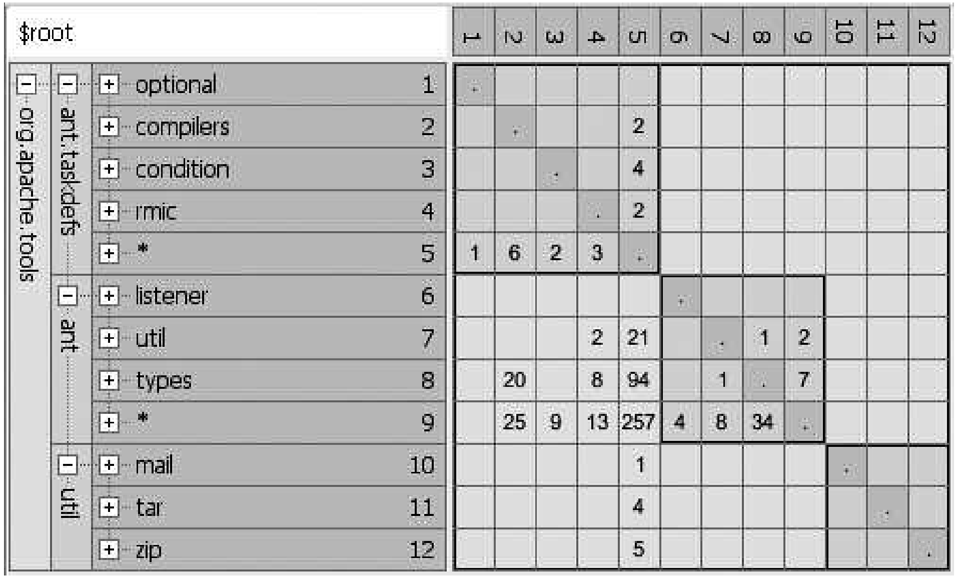
Software degrades from release to release because implicit design rules such as layering are violated. Previously, architects did not have a way to specify, track, or enforce these implicit rules. Dependency models now offer the potential for maintaining the architecture over successive revisions of the life cycle by specifying rules explicitly that define the acceptable and unacceptable dependencies between subsystems. In cases where architecture has evolved and design rules need to be changed, violations can actually make architectural evolution explicit for the entire development team

Tracking Architecture Evolution in a Dependency Model

We will apply the dependency model approach to Ant, a popular Java build utility. Ant is an open source application that is used by developers to compile, create jar files, build executables, and other sundry development tasks.

One of the principal reasons for the success of Ant has been the notion of tasks. Tasks are named entities that do the actual work and utilize the common Ant framework for doing so. Examples of tasks are *compile*, *copy*, *make directory*, and *create jar file*. Over time, a large number of tasks have been created by independent developers all over the world. This has worked well because the architects of Ant had the foresight to keep the tasks in a layer separate from the framework.

Figure 5 illustrates this conceptual architecture. Ant has been decomposed into three layers: *taskdefs*, *ant*, and *util*. The *taskdefs* layer contains the tasks, the *ant* layer contains the framework, and the *util* layer contains utilities that could be used by both the framework and tasks.



$root				1	2	3	4	5	6	7	8	9	10	11	12
org.apache.tools	ant.taskdefs	optional	1	.											
		compilers	2		.			2							
		condition	3			.		4							
		rmic	4				.	2							
		*	5	1	6	2	3	.							
	ant	listener	6						.						
		util	7				2	21		.	1	2			
		types	8		20		8	94		1	.	7			
		*	9		25	9	13	257	4	8	34	.			
	util	mail	10					1					.		
		tar	11					4						.	
		zip	12					5							.

Figure 6: *A Dependency Model for Ant Vers. 1.4.1*

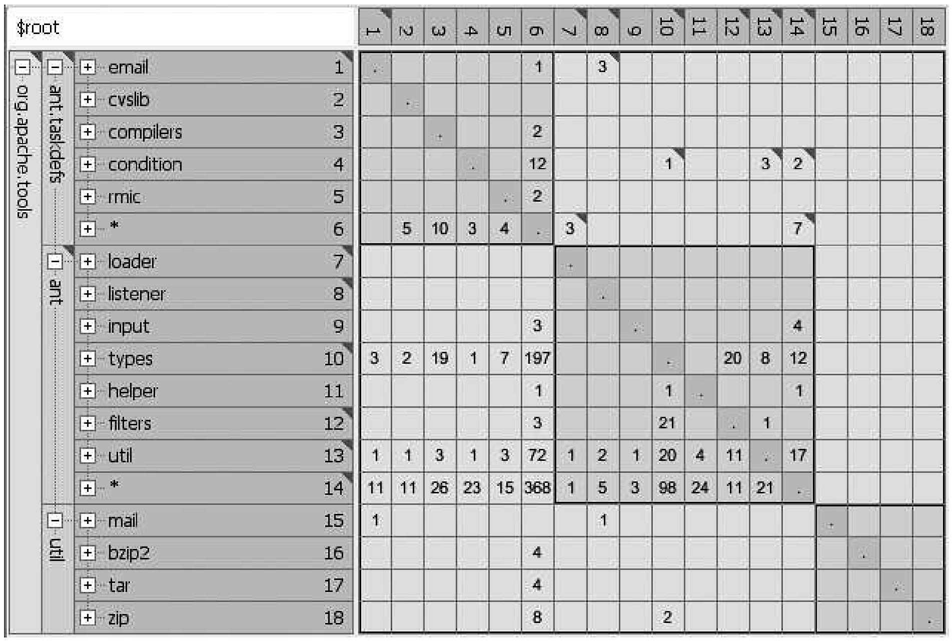
The dotted lines between the subsystems illustrate allowed dependencies, while the dotted lines with a disallow symbol illustrate disallowed dependencies. The layered approach reduces complexity and minimizes the likelihood that a bug introduced in the development of a task will affect the framework, that in turn could affect other tasks.

The DSM for Ant Vers. 1.4.1 in Figure 6 shows that Ant has three distinct layers. However, the Ant framework represented by the middle subsystem is largely monolithic. For Vers. 1.4.1, it was not a significant issue because the Ant framework was quite small at the time. We also note that the Java package names do not reflect the layering.

By comparison, the DSM for the current version of Ant in Figure 7 shows that architectural violations have begun to creep in as the Ant framework has become dependent on the taskdefs layer. Further, the application is now considerably larger while the Ant framework continues to be largely monolithic. However, it should be noted that the architecture of Ant is still not as bad as we have seen in many other commercial systems that receive less scrutiny than Ant.

Frequently, systems become so complex that development teams suggest a rewrite. This is inherently a high-risk decision since there is no way to guarantee that the new system will not suffer from similar problems of complexity. The dependency model already gives us guidance about which dependencies to fix and their relative priority. This analysis of the

Figure 7: *Dependency Model for Ant Version 1.6.1*



$root				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
org.apache.tools	ant.taskdefs	email	1	.					1		3										
		cvslib	2		.																
		compilers	3			.			2												
		condition	4				.		12				1			3	2				
		rmic	5					.	2												
		*	6		5	10	3	4	.	3							7				
	ant	loader	7							.											
		listener	8								.										
		input	9						3			.					4				
		types	10	3	2	19	1	7	197				.		20	8	12				
		helper	11						1				1	.			1				
		filters	12						3				21		.	1					
		util	13	1	1	3	1	3	72	1	2	1	20	4	11	.	17				
		*	14	11	11	26	23	15	368	1	5	3	98	24	11	21	.				
	util	mail	15	1							1							.			
		bzip2	16						4										.		
		tar	17						4											.	
		zip	18						8				2								.

dependencies reveals where the architecture is really broken and whether remediation requires a complete rewrite or just a fix to the problematic dependencies. Note that additional analysis using the DSM can easily be conducted to explore alternate organization of the architecture, and how the framework itself can be split up so it is no longer monolithic.

Dependency models can also be extended to specify the dependencies of each subsystem upon external libraries. First, the dependency model is examined to see what external libraries are used and which subsystems use them. For instance, in Ant Vers. 1.6.1, we find that some of the external libraries in use include *org.apache.bcel*, *org.apache.bsf*, and *org.apache.xml*. Design rules can then be used to specify the subsystems that are allowed to use these external libraries. Controlling the use of external libraries can pay rich dividends. In addition to helping to maintain architectural integrity, segregating the use of external libraries can also ease the job of migrating to new technologies as they become available.

Architecture Management as Part of CMMI

The dependency model provides a new way to define and enforce architecture for software engineering process initiatives such as Capability Maturity Model® Integration (CMMI®). A shared understanding of the architecture and managing its change can significantly improve the results that can be derived from these CMMI specific process areas:

- **Requirements Development:** Baseline conceptual architecture and dependency model for inclusion in the technical data package.
- **Technical Solution:** Assess architectural alternatives, formalize precise subsystem decomposition, and define the dependencies between the decomposed subsystems. Measure and verify architectural conformance.
- **Product Integration:** Identify components for product line architectures. Remove redundancies. Improve testability of products.
- **Project Planning:** Estimate impact of new feature requests and track progress against architectural changes.
- **Decision Analysis and Resolution:** Document rationale for architectural changes, understand risks, evaluate, and estimate impact of proposed changes.

® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

In addition, the dependency model approach provides new means to make architecture management part of the CMMI generic processes. It can be used to manage architecture configurations; involve stakeholders; and to evaluate, monitor, and control the architecture.

Conclusion

Architecture is integral to software quality. Unless the architecture is explicitly defined, communicated, and controlled, it will degrade as the complexity increases through the life cycle. The dependency model is a powerful new way to manage the architecture of software applications.

It is highly advisable that architecture be tested automatically as part of regular builds. Inadvertent violations can then be fixed immediately, avoiding expensive remediation later in the development cycle. This approach is lightweight – dependency models can be checked and updated automatically with intervention being required only when violations are detected.◆

References

1. Eppinger, Steven D. "Innovation at the Speed of Information." Harvard Business Review Jan. 2001.
2. Baldwin, C.Y., and K.B. Clark. The Power of Modularity Vol. 1. Cambridge, MA: MIT Press, 2000.
3. Sangal, Neeraj, Ev Jordan, Vineet Sinha, and Daniel Jackson, "Using Dependency Models to Manage Complex Software Architecture." OOPSLA 2005, San Diego, CA. 16-20 Oct. 2005.
4. Parnas, D.L. "Designing Software for Ease of Extension and Contraction." IEEE Transaction on Software Engineering 5.1 (Mar. 1979): 128-138.
5. Sullivan, K., Y. Cai, B. Hallen, and W. Griswold. "The Structure and Value of Modularity in Software Design." Proc. of the 8th European Software Engineering Conference, Vienna, Austria. 10-14 Sept. 2001.
6. Lopes, Cristina Videira, and Sushil Bajracharya. "An Analysis of Modularity in Aspect-Oriented Design." Proc. of Aspect-Oriented Software Development Conference, Chicago, IL. Mar. 2005.
7. MacCormack, Alan, John Rusnak, and Carliss Baldwin. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code." Harvard Business School. Working Paper No. 05-016.

About the Authors

Neeraj Sangal is president of Lattix Inc., which specializes in software architecture management solutions and services. He has analyzed many large proprietary and open source systems. Previously, Sangal was president of Tendril Software that pioneered model-driven Enterprise Java Beans development and synchronized Unified Modeling Language models for Java. Prior to Tendril, he managed a distributed development organization at Hewlett Packard. Sangal has published and presented papers, most recently a joint work on architecture management presented at the Object-Oriented Programming, Systems, Languages, and Applications 2005 conference.

Lattix, Inc.
8 Harper CIR
Andover, MA 01810
Phone: (978) 474-5022
E-mail: neeraj.sangal@lattix.com

Frank Waldman is vice president at Lattix, Inc. He has extensive experience building companies with innovative technology in a number of industries, including engineering software, consumer electronics, manufacturing, and product development services. Prior to Lattix, Waldman was responsible globally for building markets for the product lifecycle management software business of Eigner, which was acquired by Agile Software to create the largest pure-play product lifecycle management vendor in the global market. He has a Bachelor of Science and Master of Science from Massachusetts Institute of Technology, and holds numerous patents.

Lattix, Inc.
8 Harper CIR
Andover, MA 01810
Phone: (978) 474-5022
E-mail: frank.waldman@lattix.com

UML Design and Auto-Generated Code: Issues and Practical Solutions

Ilya Lipkin and Dr. A. Kris Huber
Hill Air Force Base

This article presents issues encountered as well as practical solutions to using the Unified Modeling Language (UML) for design and automatic code generation. Topics presented include general issues found with UML on a real-time systems design project.

This article is based on experience gained during the early history of a project being worked on at Hill Air Force Base, Utah. One of the customer requirements on this project was a specific development tool based on Unified Modeling Language (UML) Version 1.3, namely Rational Rose RealTime (RoseRT).

The project issues and solutions presented in this article are from the real-time control system. The configured software items consist of software design elements expressed in UML from which C++ code can be automatically generated. The observations presented in this article do not necessarily apply to all UML-based development tools, but the authors have made an attempt to raise a few issues of general interest to those involved in similar projects.

Overview of UML

UML is a modeling language developed by Grady Booch, James Rumbaugh, and Ivar Jacobson, and many other contributors to the Object Management Group (OMG). The focus of UML is to model systems using object-oriented concepts and methodology. UML consists of a set of model elements that standardize the design description. These elements include a number of fundamental model elements and modeling concepts, in addition to views that allow designers to examine a design from different perspectives, and diagrams to illustrate the relationships among model elements.

Several views such as Use-Case View, Logical View, Component View, Concurrency View, and Deployment View create a complete description of the system design. Within each view, an organized set of diagrams and other model elements are visible. Diagrams include use-case diagrams, class diagrams, object diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. Some key primitive model elements are states, transitions, messages, classes, class roles, attributes, and operations [1].

The UML language is complete enough that it offers the exciting ability to allow the creation of auto-generated code that implements the design. The code can be generated from the system description of the model through the use of diagrams and other model elements.

UML Elements for Real-Time Systems Design

Designing real-time systems is challenging. In UML, an *active class* model element was introduced to address this challenge. The purpose of this element was to help simplify both the design and the implementation.

The active class model element consists of a communication structure description and a behavioral description. The communication structure is described using a collaboration diagram that shows the ports through which it sends and receives messages to and from other active classes. The behavior is described using a statechart diagram that shows how the active class acts and reacts to its environment[1]. In other words, the active class is a standalone *capsule* of software that talks to its environment through ports (specified in the structure diagram), and performs *actions* as it transitions through a sequence of states (specified by the statechart diagram).

The characteristics of a run-time system (RTS) object and the UML active class were determined to simplify the process of real-time software design and implementation. In addition, by encapsulating calls to the operating system of the target platform within the RTS, the auto-generated implementation of the UML design can be made largely platform-independent. Real-time application design with UML is discussed in [2].

UML Issues and Practical Solutions

While working on the project, several problems dealing with UML for design and implementation were encountered. Below are some of the issues related to UML design and implementation. These issues, some solutions, and open questions will be presented using small illustrations.

Issue No. 1: State Diagram Clutter

When designing with UML, it is easy for a developer to put too much information into a single diagram. When this happens, information overload will make the design difficult to understand and maintain.

Example

It is possible to describe the behavior of an entire design in a single UML statechart diagram. To demonstrate the potential for diagram clutter, let us consider this simple case: There is a system model that will take user input, button press (an event), and convert it to a textual presentation of the button press to be displayed on screen. An all-UML solution for this simple case is shown in Figure 1.

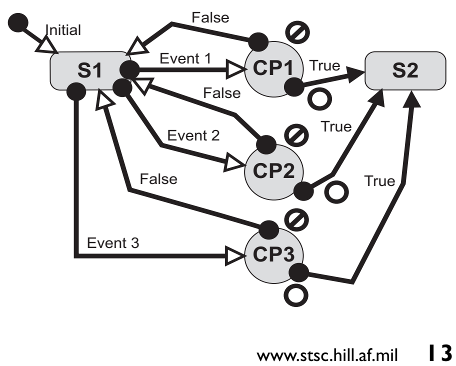
The design consists of a set of three choice points and two states with transitions between them. Each choice point represents a decision to be made when an event-triggered transition has occurred. State S1 represents the entry into the system model and S2 is the final state of the system model.

Since this system model is designed with RoseRT UML elements, the state machine solution is a fully functional implementation and design. For implementation, C++ is auto-generated from the UML design. This simple design solution, although straightforward, is somewhat hard to understand from the statechart diagram in Figure 1. This is because the view of the system is cluttered by transition labels and a large number of UML elements presented in a small visual area.

Practical Solutions

In Figure 2 (see page 14), the solution to the same problem is shown, but utilizing a mixed UML and C++ approach to simplify the model. The *action code* in the transition *Event* would perform the logic implemented by the choice points (circles) in Figure 1, as

Figure 1: *All-UML Solution*



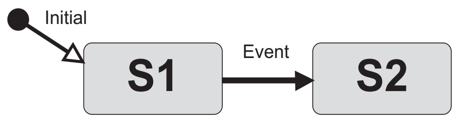


Figure 2: *Combined UML/C++ Solution*

illustrated by the following pseudocode:

```
if(Event1) then
        display string1
else if(Event2) then
        display string2
else if(Event3) then
        display string3
```

This solution moves logic that is straightforward and redundant into the state transition between states S1 and S2. In addition, it also keeps the same overall logic of the UML statechart of Figure 1.

Another benefit to the UML and C++ solution is that the size of the system model can be reduced. The Figure 1 solution produces an implementation consisting of 315 lines of auto-generated code and three lines of *user code* (code that is inserted by the user and copied verbatim by the code generator). On the other hand, the mixed UML/C++ solution in Figure 2 produces an implementation consisting of 215 lines of auto-generated code and 14 lines of user C++ code.

This example demonstrates that the size of the auto-generated code can be reduced through using a mixed UML/C++ design approach. The code efficiency and maintainability also seem to be improved in this case, although this will not be true for all designs. The maintainability of a UML-based software object will depend upon several factors, including the maintenance team's relative proficiency with UML compared to the underlying implementation language.

An alternate method of reducing the number of visual elements on a model diagram is to design hierarchical state machines. If hierarchal statecharts are supported by the design tool, multiple states can be combined to create a clearer, high-level statechart containing fewer states. This approach to reducing visual elements has an additional benefit of increased portability of the design, but may not achieve the implementation efficiency of the combined UML/C++ approach.

Guideline

Although UML is very flexible, some implementation and design considerations should be addressed through conventional means by either manually coding or using system flow diagrams outside of UML [3].

It is important to keep the number of states and transitions to a minimum. As the model complexity increases, understanding of the design can be decreased due to clutter in the design diagrams. The efficiency of the auto-generated implementation of the design can also be reduced. Depending upon the need for implementation efficiency, and the team member familiarity with UML, an appropriate design philosophy should be established for the project.

Issue No. 2: UML Metrics

The authors were not aware of any useful software size metrics that could be used with UML models. Using the source lines of code (SLOC) metric on auto-generated code tended to overemphasize the UML portion of a mixed UML/C++ model (with the UML portion being auto generated and the C++ portion being user code). Metrics were needed both for project estimation and project tracking.

Discussion

Many design rules have been proposed for judging UML design quality [4]. However, it is hard to know how to track effort used on developing software using such metrics. Obviously, SLOC for auto-generated code size could be misleading, but to be feasible, the project tracking and estimation tools often require using a scalar metric like SLOC rather than a vector of metrics. The challenge is to correlate the size metric to the amount of development effort required for design and implementation of the software. Experimentation with the SLOC metric upon auto-generated code seemed to overemphasize the UML portion of a mixed UML/C++ model.

Example

How do we judge the total size of the models of Figures 1 and 2?

Practical Solution

To arrive at the solution presented below, it was necessary to experiment with more than one metric for total size. One of the possible methods is illustrated in the solution of issue No. 1: Measure the auto-generated code separately from the user code that is manually maintained. This approach is adequate for some purposes, but in other situations, a metric is needed that captures the size in a single number.

To remedy shortfalls of the previous approach, a second metric was defined that consists of a single number to represent model size. A utility tool was developed that could traverse the model and provide a total count of several of the key UML model elements. Based on counts of model elements and user code SLOC, this metric could be calculated as a weighted sum. The weights shown in the second column of Table 1 were used to give a size estimate in terms of implementation units (IU).

Using this UML metric, the size of the models in Figures 1 and 2 are calculated from the weighted sum of their model elements to be 107 and 52 IU, respectively. The weights associated with this metric were estimated subjectively based on the RoseRT user interface for each of the model elements. The resulting IU metric has a level of detail similar to SLOC, and the measure may be thought of as a SLOC-equivalent metric. The weights associated with this metric have yet to be fully substantiated due to insufficient historical data for the project. For this reason, project history data is maintained in a raw format so that new weights (and even new metrics) can be applied to the entire project history.

This second metric for estimating the size of a combined UML/C++ model/implementation is similar in spirit to the function points approach. In place of logical or functional elements, UML model elements were given SLOC-equivalent weights.

Guideline

It is best to find a metric that correlates well to the quantity desired. Employ UML design tools that contain rich application programming interfaces (APIs) to facilitate development of utilities that automate the process of metrics calculation. Collecting data in a raw format can facilitate adjustment or customization of the metrics.

Table 1: *Weights and Counts for Key UML Model Elements*

Model Element	Weight (IU)	Counts for Figure 1	Counts for Figure 2
States^2	6	5	2
Transitions	6	10	2
Capsule Roles	9	1	1
Ports	5	1	1
Attributes	1	0	0
Operations	4	0	0
Signals	2	0	0
User Code	1	3	14
TOTAL (IU)		**107**	**52**

Issue No. 3: Documentation for Graphical Elements

Although UML implements graphical methods of presenting software solutions, it is very challenging to create a self-documenting model. Traditional documentation of the graphical model is still needed.

Discussion

Many developers consider documentation to be the least favorable task of any new design. UML, combined with automatic report generation capability in the tool, allows the possibility of design and implementation to be self-documenting. When using UML for the design task, the designer must create a set of UML diagrams and other graphical information such as states, transitions, and illustrations of the structured relationship among objects. The final product is a model that contains design information in a graphical format that should be suitable for documentation. Ideally if the design is well drawn, then graphical information is actually usable for documentation, and a lot of time and effort savings can be realized.

Example

Figure 1 is an example of a UML model that is not self-documenting. Looking at this figure, several questions can be raised. What do choice points do? What is being compared? What do *true* and *false* labels mean? What do State S1 and State S2 represent? The answer to these questions is that S1 is an initial state and S2 is the final state. The choice points are the logic to set the string to be displayed based on events that occur. *True* and *false* label the condition-dependent transitions from the choice points.

UML tools can produce a sequence diagram from a statechart diagram, as well as generate a number of documentation reports. These typically exclude any user code implementation documentation. Project experience found customization of documentation reports to be non-trivial.

Practical Solution

In the example of Figure 1, a set of supporting documentation is still necessary to accompany a graphical design. Although this seems to be a very obvious observation, the project currently being worked on had omitted to document the graphical design. As a result, the loss in ability to understand and maintain statecharts had increased cost and delayed schedule. In addition, maintainability of the real-time system had been drastically reduced. Although prudent choice of names for the model elements can help make the diagrams more self-documenting, the use of UML use cases and their relationship to the graphical state-chart solutions must be documented rather than assumed to be self-documenting. Software developers must come up with a set of documentation that bridges the link between use cases and their UML graphical representations. The final gap to be filled is the augmentation of documentation generated automatically from UML with manual description for user code and how it fits with design implementation.

Guideline

Although UML presents design visually, comments on the states or any other model elements are still needed. These comments clarify the overall solution and explain some of the design choices of the model diagram [5]. Self-documenting features of UML are still no substitute for design notes or additional supporting documentation on design decisions.

Issue No. 4: Design Portability Between UML Tools (Open Question)

Using custom (i.e., nonstandard) features of UML tools reduces portability of a design to other UML-compliant development tools.

Example

RoseRT had introduced additional concepts into UML to accommodate the needs of the real-time environment, and to allow for a development tool to produce auto-generated code solutions. One of the new concepts to UML introduced by RoseRT was a structure diagram, Figure 3. Structure diagrams are used in RoseRT to link capsules together in a coherent way.

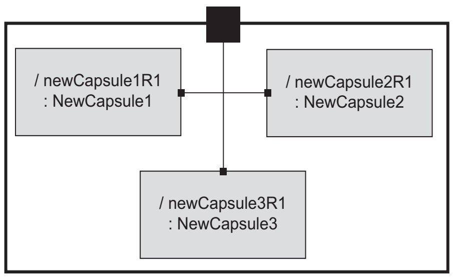


Figure 3: *Structure Diagram*

Practical Solution

Unfortunately, there is no straightforward solution for this problem; the only thing that can be used is a set of mitigation strategies. One of those mitigation strategies was the choice of product itself. Although this was done at the program office, the UML implementation chosen was from the primary contributors to the UML standard. Structure diagrams were not part of UML initially, but rather RoseRT's structure diagrams were based on the UML 1.3 collaboration diagram. Another notable difference is the notion of capsules, which are presented in Figures 1 and 2. Capsules are, in essence, UML 1.3 active classes and will be part of UML 2.0 as *structured classes*.

Another strategy is to avoid external API calls in the user code portion of the model, as it will make the design less portable to other UML solutions. Also, the RoseRT UML tools come with a set of custom API calls that can be used to enhance design and development efforts, but it is best to avoid using them if design portability is an issue.

Guideline

Some of the UML development tools offer specialized controls or functionality that do not exist in other tools. It is a good idea to explore means for design using standardized UML components rather than customized ones. This requires the design team to become familiar with the UML standards to recognize and avoid nonstandard items, thereby providing portability across development tool vendors.

Issue No. 5: Cross Language and Cross Platform Development

Target languages or their compilers can have platform-specific features that defeat the portability of a mixed UML/user-code solution.

Discussion

One of the advantages of UML development is that it is language-neutral, and platform-independent. Using UML, it is possible to disconnect the design from possible issues of implementation that are based on the choice of source code language. In addition, the ability to take UML design as is, and to auto-generate source code from it, allows for the creation of implementation directly from the design [6]. Due to UML being language neutral, the solution will not change if the target language is C/C++, Java, or anything else [7]. In addition, if one of the requirements is to maintain the same code in two languages, it becomes simpler to have bug reports and fixes done in one place rather then two.

The ability to create a design that is platform-independent provides a set of unique opportunities and challenges. Cost may be reduced by enabling a portion of testing to take place on a development platform that does not necessarily include a simulation of the target environment. Multi-platform testing may have a beneficial effect of finding some defects that would otherwise be masked on one of the individual platforms. The challenge of platform-independent design is to ensure that special requirements of the final target environment are

State No., Name	Description	Attributes
Initial	System not running	Timer event set to 30 seconds
Green	On timer time-out event go to(goYellow) Yellow	Timer event set to 45 seconds
Yellow	On timer time-out event go to(goRed) Red	Timer event set to 10 seconds
Red	On timer time-out event go to(goGreen) Green	Timer event set to 30 seconds

Table 2: *State Specification Template*

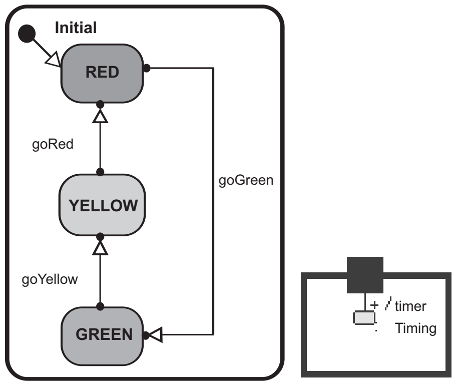


Figure 4: *Working UML Design and Implementation Solution for Traffic Light*

being considered in the design of the user code portion of the UML model.

Example

A traffic light design scenario can be used to demonstrate the ability to have a cross-language and cross-platform development. This small UML design example shows both design and implementation.

Requirements for traffic light are presented below. The traffic light has three colors: red, yellow, and green. To change from green to yellow, an event of time expiring is needed. The events always happen in a fixed sequence of green to yellow to red. It is not allowed to go from green to red, for example. The State Specification Template (Table 2) is used to describe a sequence of state-change events [8]. Timer events are written as generic RoseRT descriptions for each event as "timer.informIn(timeout);" this is the only line of code that is written for the entire design and implementation.

Practical Solution

The UML solution in Figure 4 is completely language- and platform-neutral. There is no user code associated with it that is language- or platform-specific. Therefore it is possible for an auto-code generation engine to translate the UML to a destination language or platform of choice.

Guideline

When implementing UML design, it is best to avoid considerations of how code generation will translate UML to a target language or system platform for the auto-generated portion of the solution. However, when entering user code into mixed UML models, extra care must be taken to avoid a platform-specific syntax in the implementation language.

Conclusion

UML design introduces new and unexplored paradigms that can either simplify the task or make it overly complex. It is important to adhere to proven methods and concepts to utilize what is available. Practical solutions and guidelines related to diagram clutter, UML metrics, documentation, design portability, and cross-language/cross-platform development presented in this article are the tip of an iceberg in a great sea of development.◆

References

1. Sanderfer, Lynn. "How and Why to Use the Unified Modeling Language." CrossTalk, June 2005 <www.stsc.hill.af.mil/crosstalk/2005/06/0506Sanderfer.html>.
2. Gomaa, Hassan. Designing Concurrent, Distributed, and Real-Time Applications With UML. Addison-Wesley, 2000.
3. Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Design and the Unified Process. 2nd ed. Prentice-Hall, 2002.
4. Wüst, Jürgen. SDMetrics 23 Apr. 2005 <www.sdmetrics.com/LoR.html>.
5. Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Addison-Wesley, 2004.
6. Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. 2nd ed. Addison-Wesley, 2005.
7. Satzinger, John W., Robert B. Jackson, and Stephen D. Burd. Object-Oriented Analysis and Design with the Unified Process. Thomson, 2005.
8. Humphrey, Watts S. A Discipline for Software Engineering. Addison-Wesley, 1995.

Notes

1. In the RoseRT tool, active classes are called capsules; the associated collaboration diagrams are called *structure* diagrams.
2. Choice points are considered *pseudostates* that are counted as states when calculating this metric.

About the Authors

Ilya Lipkin is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah. His current research interests include artificial intelligence, human knowledge capture and analysis, neural networks, fuzzy logic, user interface design, software engineering, and customer relations management. Lipkin has a Bachelor of Science in computer engineering from the University of Toledo, a Master of Science in computer engineering from the University of Michigan, and is a doctoral candidate at the University of Toledo Business School.

309 SMXG/MXDEE
7278 4th ST BLDG 100
Hill AFB, UT 84056
Phone: (801) 586-4477
Fax: (801) 586-2042
E-mail: ilya.lipkin@hill.af.mil

A. Kris Huber, Ph.D., is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah, where he has been working for two years on an embedded software engineering project. Previously, he worked on video compression algorithm research, development, and MPEG-4 standardization for Sorenson Media. His interests are software engineering, computers, and electronic systems. Huber has a Bachelor of Science in electrical engineering from Brigham Young University, and master's and doctorate degrees in electrical engineering from Utah State University.

309 SMXG/MXDEE
7278 4th ST BLDG 100
Hill AFB, UT 84056
Phone: (801) 586-5535
Fax: (801) 586-2042
E-mail: kris.huber@hill.af.mil

UML 2.0-Based Systems Engineering Using a Model-Driven Development Approach

Dr. Hans-Peter Hoffmann
I-Logix Inc.

More and more, systems engineers are turning to the Unified Modeling Language (UML) to specify and structure their systems. This has many advantages, including verifiability and ease of passing off information to other engineering disciplines, particularly software. This article describes a UML 2.0-based process that systems engineers can use to capture requirements and specify architecture. The process uses the UML exclusively for the representation and specification of system characteristics. Essential UML artifacts include use-case diagrams, sequence diagrams, activity diagrams, state-chart diagrams, and structure diagrams. The process is function-driven and is based heavily on the identification and elaboration of operational contracts, a message-based interface communication concept. The outlined process has been applied successfully at various customer sites. It is assumed that the reader is familiar with the basics of UML.

For many years, software engineers have successfully applied the Unified Modeling Language (UML) to model-based software engineering. There have been several attempts to introduce UML and the underlying object-oriented methods to systems engineering to unify the overall development process. However, many systems engineers continue to use classically structured analysis techniques and artifacts to define system requirements and designs.

One reason for this could be that systems engineering is mostly driven by functional requirements. Speaking in terms of system functionality is still considered the most natural way of expressing a design by most of the domain experts involved in the early phases of system development (electrical engineers, mechanical engineers, test engineers, marketing personnel, and, of course, customers). Given this, the only way to unify the overall development process is to extend UML with regard to function-driven systems engineering, and to define a process that enables a seamless transfer of respective artifacts to UML-based software engineering. The release of UML 2.0 [1] eventually provided the missing artifacts for function-driven systems engineering.

The following sections describe a UML 2.0-based process that systems engineers can use to capture requirements and specify architecture. The approach uses model execution as a means for requirements verification and validation.

Process Overview

Figure 1 shows the integrated systems and software engineering process by means of the classic *V*. The left leg of the V describes the top-down design flow, while the right-hand side shows the bottom-up integration phases from unit test to the final system acceptance. Using the notation of state-charts, the iterative characteristic of the process is visualized by the *high-level interrupt* due to system changes. Whenever changes occur, the process will restart at the requirements analysis phase.

Systems engineering is characterized by a sequential top-down workflow from requirements analysis to system analysis and system architectural design. A loop back to the requirements analysis may occur if, during systems analysis or architectural design, functional issues require a reexamination of the higher-level requirements. The software engineering workflow is characterized by the iterative and incremental cycles through the software analysis and design phase, the implementation phase, and the different levels of integration and testing.

It is important to note the creation and reuse of requirements-related test scenarios along the top-down design path. These scenarios are also used to assist the bottom-up integration and test phases and, in the case of system changes, regression test cycles.

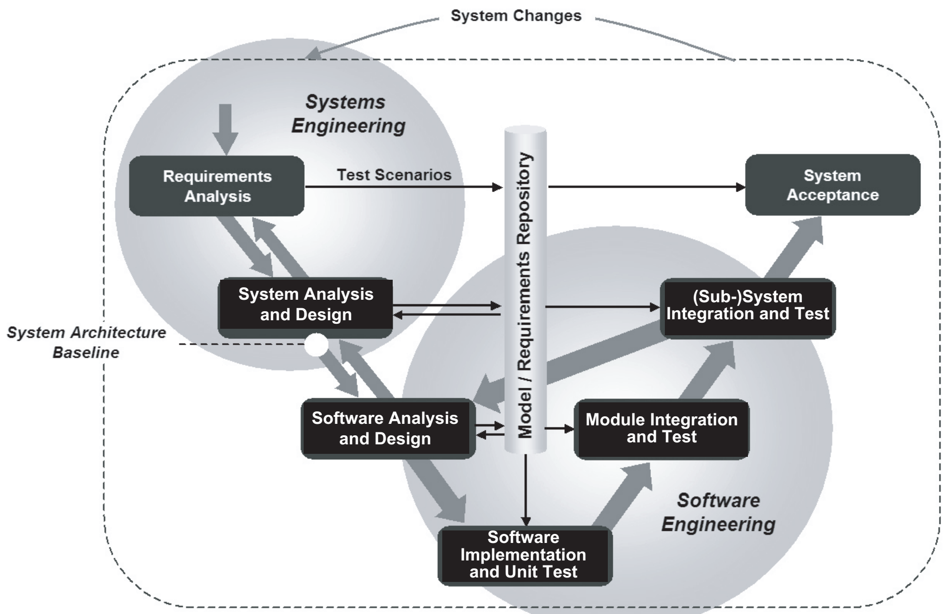
Key objectives of the systems engineering process are as follows:

- Identification and derivation of required system functionality.
- Identification of associated system states and modes.
- Allocation of system functionality and modes to a physical architecture.

Regarding modeling, these key objectives imply a top-down approach on a high level of abstraction. The main emphasis is on the identification and allocation of a needed functionality (e.g., a target tracker), rather than on the details of its behavior (e.g., the tracking algorithm).

Figure 2 depicts an overview of the UML-based systems engineering process. For each of the systems engineering phases, it shows the essential tasks, associated input, and work products.

Figure 1: *The Integrated Systems and Software Engineering Process*


System Changes
Systems Engineering
Requirements Analysis
Test Scenarios
Model / Requirements Repository
System Acceptance
System Analysis and Design
(Sub-)System Integration and Test
System Architecture Baseline
Software Analysis and Design
Module Integration and Test
Software Engineering
Software Implementation and Unit Test

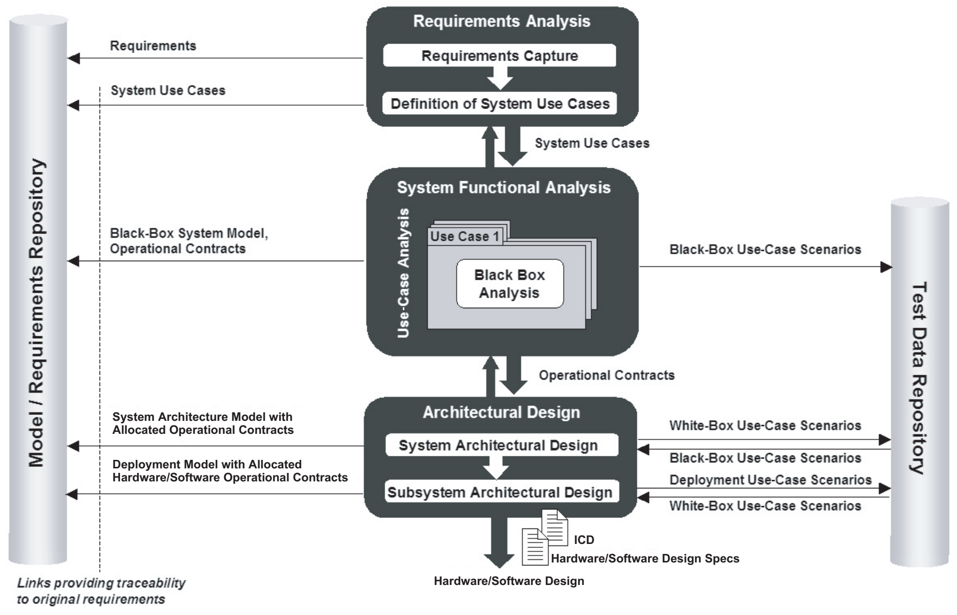

Requirements Analysis
Requirements Capture
Definition of System Use Cases
System Use Cases
System Functional Analysis
Use-Case Analysis
Use Case 1
Black Box Analysis
Operational Contracts
Architectural Design
System Architectural Design
Subsystem Architectural Design
ICD
Hardware/Software Design Specs
Hardware/Software Design
Model / Requirements Repository
Test Data Repository
Requirements
System Use Cases
Black-Box System Model, Operational Contracts
System Architecture Model with Allocated Operational Contracts
Deployment Model with Allocated Hardware/Software Operational Contracts
Black-Box Use-Case Scenarios
White-Box Use-Case Scenarios
Black-Box Use-Case Scenarios
Deployment Use-Case Scenarios
White-Box Use-Case Scenarios
Links providing traceability to original requirements

Figure 2: *UML-Based Systems Engineering Process*

The process is operational contract-driven. An operational contract specifies system behavior by adding pre- and post-conditions to the description of respective operations. Operational contracts are the primary source of traceability to the initial requirements.

The essential tasks in the requirements analysis phase are requirements capture and the grouping of requirements into use cases.

The main emphasis of the system functional analysis phase is on the transformation of the identified functional requirements into a coherent description of system functions (operational contracts). Each use case is translated into a respective black-box model and verified and validated through model execution. Incrementally, these black-box use-case models are merged to an overall black-box system model.

The focus of the subsequent system architectural design phase is the allocation of the verified and validated operational contracts to a physical architecture. The allocation is an iterative process. In collaboration with domain experts, different architectural concepts and allocation strategies may be analyzed, taking into consideration performance and safety requirements that were captured during the requirements analysis phase.

In the subsequent subsystem architectural design phase, decisions are made on which operational contracts within a physical subsystem should be implemented in hardware and which should be implemented in software (hardware/ software trade-off analysis). The different design concepts are captured in the deployment model and verified through regression testing.

The deployment model defines the system architecture baseline for the subsequent hardware/software (HW/SW) development. Essential documents that are generated from the deployment model are the following:

- HW/SW design specifications.
- Logical interface control document (ICD).

The outlined systems engineering process is model-based, using the UML 2.0 as a modeling language. The essential UML artifacts are the following:

- Use-case diagram.
- Activity diagram.
- State-chart diagram.
- Sequence diagram.
- Structure diagram.

The following sections detail the work-flow in the different systems engineering phases with examples from a case study.

Requirements Analysis

Requirements Capture

The requirements analysis phase starts with the analysis of the process inputs. Customer requirements are translated into a set of requirements that define what the system must do (functional requirements) and how well it must perform (quality of service requirements). The captured requirements are imported into the model/requirements repository.

Definition of Use Cases

Once the requirements are sufficiently understood, they are clustered in use cases. A use case describes a specific operational aspect of the system (operational thread). It specifies the behavior as perceived by the users and the message flow between the users and the use case. It does not reveal or imply the system's internal structure (black-box view).

A set of use cases is grouped in a use-case diagram. Use cases can be structured hierarchically. There is no *golden rule* with regard to the number of use cases needed to describe a system. Experience shows that for large systems, typically six to 24 use cases are defined at the top level. At the lowest level, a use case should be described by at least five with a maximum of 25 essential use-case scenarios.

At this stage, emphasis is put on the identification of *sunny day* use cases, assuming an error/fail-free system behavior. Exception scenarios are identified at a later stage (=> system functional analysis) through model execution. If more than five error/fail scenarios are found for a use case, a separate exception use case will be defined and added to the sunny day use case via the *include* or *extend* relationship.

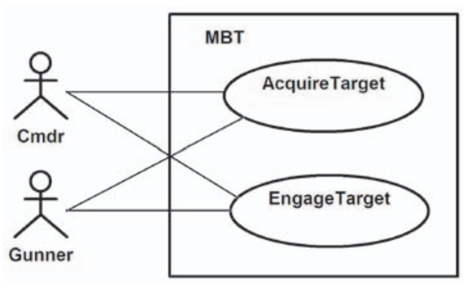
Figure 3 shows the use-case diagram of the case study. The system is a main battle tank. Two use cases are studied:

- Acquire target.
- Engage target.

The actors are commander and gunner.

The use-case diagram is imported into the model/requirements repository. The use cases are then linked to the system requirements and checked for complete coverage.

Figure 3: *Case Study Use-Case Diagram*



System Functional Analysis

In the system functional analysis phase, each use case is translated into a model and the underlying requirements then verified and validated through model execution. A message-driven approach (Figure 4) is used. Characteristics of this approach are the following:

- The system structure is described by means of a UML 2.0 structure diagram using blocks as basic structure elements and ports, and the notation of *provided/required* block interfaces.
- Communication between blocks is based on messages (*service requests*).

- System functionality is captured through operational contracts (*services*), e.g., operation1(),.., operation4() in Figure 4.
- Functional decomposition is performed through decomposition of operational contracts.

The black-box use case analysis starts with the definition of the use case model context diagram. The UML 2.0 artifact used for this is the structure diagram. Elements of this diagram are blocks representing the actors and the system.

Identifying the black-box use-case scenario is the next analysis step. A use-case scenario describes a specific path (functional flow) through a use case. It details the message flow between the actors and the use case and the resulting behavior (operational contracts) of the recipient. In the UML, a scenario is graphically represented in a sequence diagram. The lifelines in the black-box sequence diagram are the actors and the system (see Figure 5).

Once a set of essential scenarios is captured, the identified functional flow information is merged into a common use case description. The UML artifact used for this is the activity diagram (see Figure 6). Each action block in this diagram corresponds to an operational contract in a sequence diagram. The black-box activity diagram plays an essential role in the upcoming architectural design phase.

Based on the information captured in the black-box sequence diagrams and black-box activity diagram, the blocks of the black-box use case model next are populated, and ports and associated interfaces are defined. Figure 7 (see page 20) shows the resulting static model of the use case Acquire Target.

The next step in black-box use case analysis is the description of system-level, state-based behavior. The UML artifact used for this is the state-chart diagram (see Figure 8 on page 20). State-charts are hierarchical state machines that visualize system states and modes and their changes as the response to external stimuli. The use case related state-based behavior is derived from the captured use-case scenarios.

At this stage, the verification and validation (V&V) of the black-box use case model and the underlying requirements (i.e., operational contracts) can start. V&V is performed through model execution using the captured black-box use-case scenarios as the basis for respective stimuli. It should be noted that following the previously outlined key objectives of this process, the focus is on the analysis of the generated sequences rather than on the underlying functionality.

So far, the use-case model represents an error/fail free (*sunny-day*) behavior. At this stage, it may be extended regarding possible exceptions (*rainy-day* behavior).

The outlined steps are repeated for each use case, except that instead of creating new activity diagrams, the initial black-box activity diagram incrementally is extended based on the information of the new use-case scenarios. The same applies to the system black-box state-chart diagram. The subsequent model V&V is performed in two steps: First, the extended black-box system model is verified/validated through model execution using the black-box use-case scenarios as the basis for respective stimuli. Then, the collaboration of the implemented use cases is verified through regression testing.

At the end of the functional analysis phase, a black-box system model is built of verified and validated operational contracts, representing the underlying functional requirements. The black-box system model is imported into the model/requirements repository and the operational contracts are linked to the high-level system requirements. The black-box use-case scenarios are imported into the test data repository for reuse in the subsequent architectural design phases.

Architectural Design

System Architectural Design

Focus of the system architectural design phase is the allocation of the verified and validated operational contracts to a physical architecture. The allocation is an iterative process and is performed in collaboration with domain experts. Different architectural concepts and allocation strategies may be analyzed, taking into consideration performance and safety requirements that were cap-

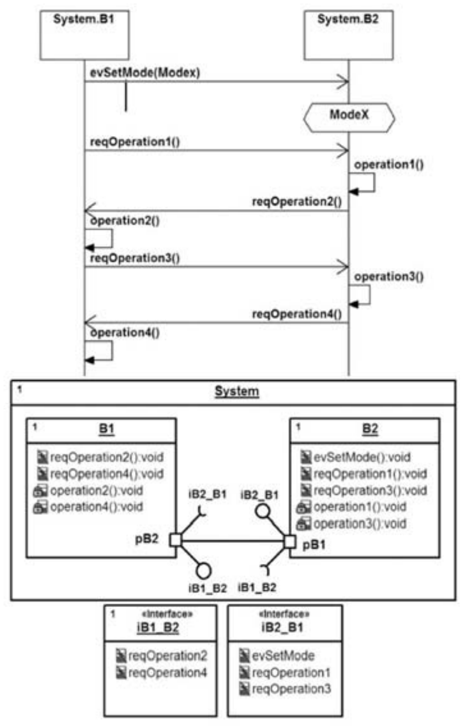
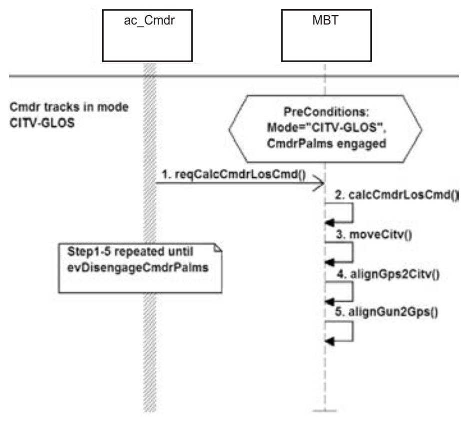


Figure 4: *Message-Driven Modeling Approach*


ac_Cmdr
MBT
Cmdr tracks in mode CITV-GLOS
PreConditions: Mode="CITV-GLOS", CmdrPalms engaged
1. reqCalcCmdrLosCmd()
2. calcCmdrLosCmd()
3. moveCitv()
4. alignGps2Citv()
5. alignGun2Gps()
Step1-5 repeated until evDisengageCmdrPalms

UC1BB21
CITV = Commander's Independent Thermal Viewer
GPS = Gunner Primary Sight

Figure 5: *Black-Box Use-Case Scenario*

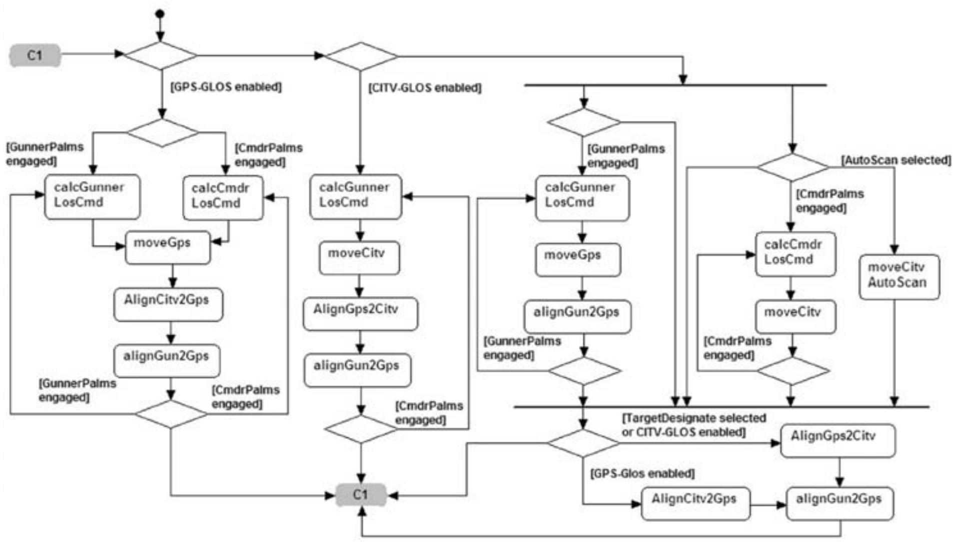


Figure 6: *Black-Box Activity Diagram (Use Case Acquire Target)*

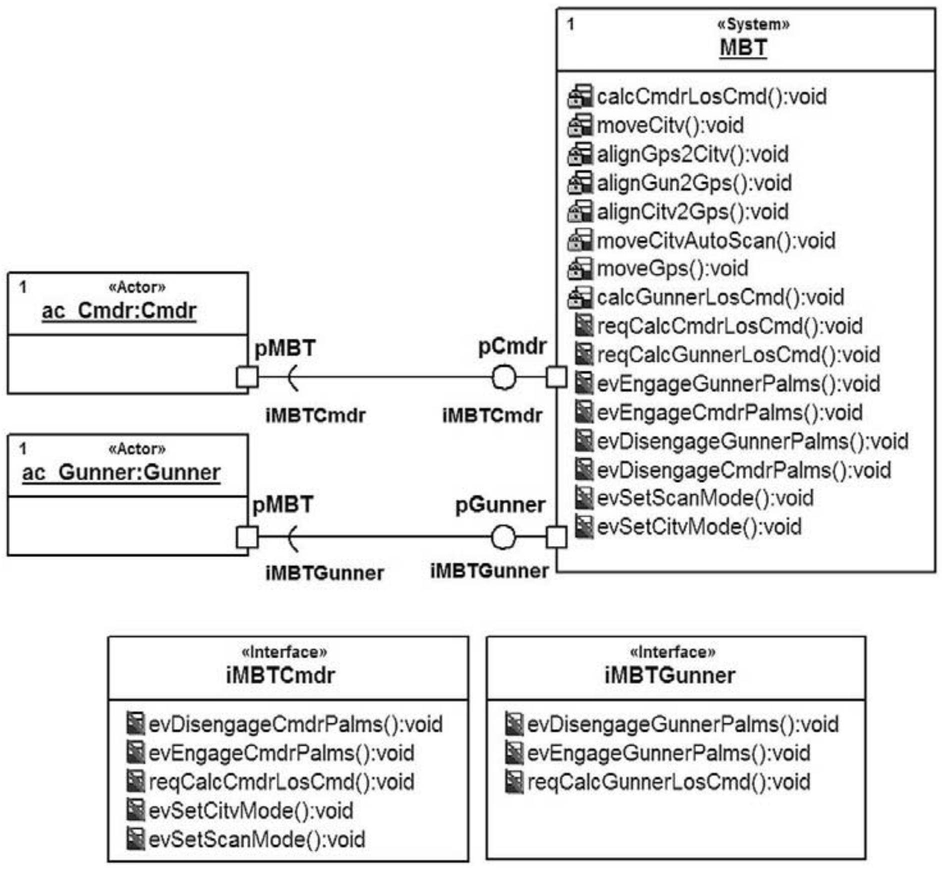


Figure 7: *Static Black-Box System Model (Use Case Acquire Target)*

tured during the requirements analysis phase.

System architectural design starts with the definition of the physical subsystems. The UML 2.0 artifact used for this is the structure diagram. Constituents of this model are the actor blocks and the system block. Parts of the system block are the physical subsystems of the chosen architecture. In the case study, the system design consists of six physical subsystems (LRU = line replaceable unit, see Figure 9A).

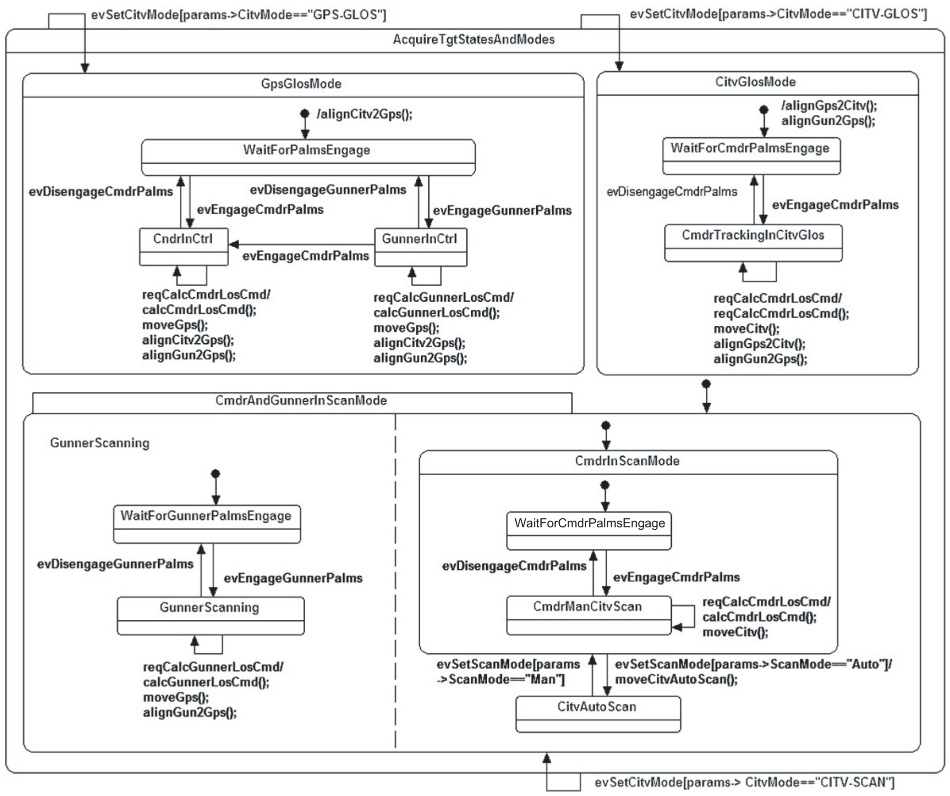
Next, the previously identified black-box operational contracts are allocated to the physical subsystems using an activity diagram (white-box activity diagram). Essentially, this activity diagram is a copy of the black-box activity diagram. The only difference is that the system now is partitioned into swim lanes, each representing a physical subsystem. Based on the chosen design concept, the system operational contracts are *moved* to respective subsystem swim lanes. An essential precondition for this allocation is that the initial links (functional flow) between the operational contracts are maintained.

The white-box activity diagram is complemented by the definition of white-box sequence diagrams. White-box sequence diagrams are decompositions of the previously captured black-box sequence diagrams and are utilized to identify the interfaces of the physical subsystems. In white-box sequence diagrams, the system lifeline is split into a set of subsystem lifelines. Based on the allocation defined in the white-box activity diagram, the subsystem operational contracts are placed on respective subsystem lifelines. To maintain the initial functional flow, service requests from one physical subsystem to the other may need to be generated. They define the interfaces between the subsystems.

In the example shown in Figure 10, the implementation concept was that LRU1 was considered the commander's input/output device. Most of the identified functionality had to be implemented in LRU5. The Commander's Independent Thermal Viewer control had to be in LRU6. In this scenario, LRU4 served as a gateway between LRU1 and LRU5. By mapping the use case scenario to the physical architecture, the links and the associated ports and interfaces are defined for each involved physical subsystem. For each physical subsystem, the associated state-based behavior is captured in a state-chart diagram. These state-chart diagrams will extend incrementally with each mapped use-case scenario.

The outlined process is performed iteratively for all black-box scenarios. Figure 9A shows the final result. Figure 9B depicts for the chosen architectural design the resulting physical subsystem interfaces by means of an N-squared (N^2) chart. An N^2 chart is structured by locating the nodes of communication on the diagonal, resulting in an NxN matrix for a set of N nodes. For a given node, all out-

Figure 8: *Black-Box System State-Based Behavior (Use Case Acquire Target)*



puts (UML 2.0 *required interfaces*) are located in a row of that node and inputs (UML 2.0 *provided interfaces*) are in the column of that node.

The correctness and completeness of the system architecture model is checked through model execution. Once the model functionality is verified, the architectural design can be analyzed with regard to the performance and safety requirements. The analysis typically includes failure modes effects analysis and mission criticality analysis.

Subsystem Architectural Design

This phase focuses on the implementation of the allocated operational contracts. Decisions are made on which operational contracts in a physical subsystem should be implemented in hardware (mechanical/application-specific integrated circuit) and which should be implemented in software. For operational contracts that span more than one domain, further analysis will be needed. Subsystem domain experts may participate in this analysis.

Once the HW/SW design decisions are made, the workflow is similar to the one outlined in the system architectural design phase. In each white-box use-case scenario, the physical subsystem lifelines are split into HW and/or SW lifelines, each representing a subsystem component. Based on the chosen HW/SW design concept, operational contracts then are placed on respective subsystem component lifelines, and the associated functional flow between subsystems and subsystem components established through respective service requests (see Figure 11 on page 22). Thus, subsystem component ports and interfaces are defined. For each physical subsystem component, the associated state-based behavior is captured in a state-chart diagram. These state-chart diagrams will extend incrementally with each white-box use-case scenario.

The outlined process is performed iteratively for all white-box scenarios. The final deployment architecture is verified through regression testing.

At the end of the system architectural design phase, the deployment model is imported into the model/requirements repository, and the HW/SW assigned operational contracts are linked to the original requirements. For each physical subsystem the following documents are generated from the deployment model as handoffs to the subsequent hardware and software design:

- HW/SW design specification.

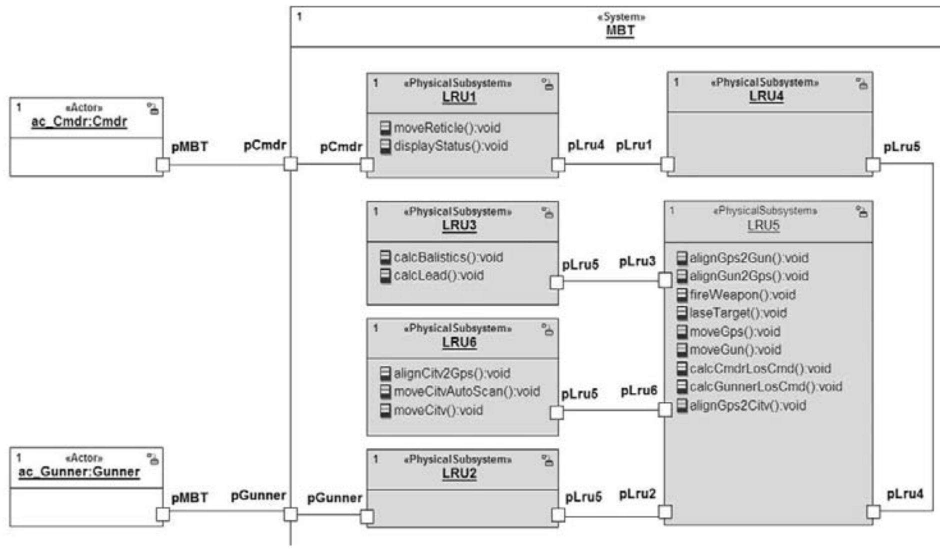
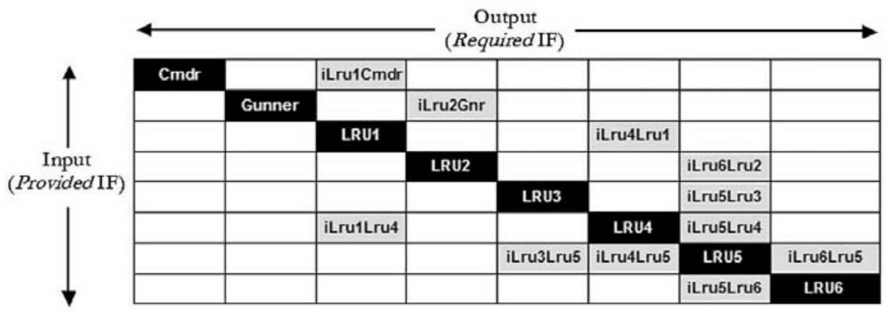


Figure 9A: *Activity Operational Contracts Allocated to an LRU Network Architecture*


Output
(*Required* IF)
Input
(*Provided* IF)

Cmdr		iLru1Cmdr					
	Gunner		iLru2Gnr				
		LRU1			iLru4Lru1		
			LRU2			iLru6Lru2	
				LRU3		iLru5Lru3	
		iLru1Lru4			LRU4	iLru5Lru4	
				iLru3Lru5	iLru4Lru5	LRU5	iLru6Lru5
						iLru5Lru6	LRU6

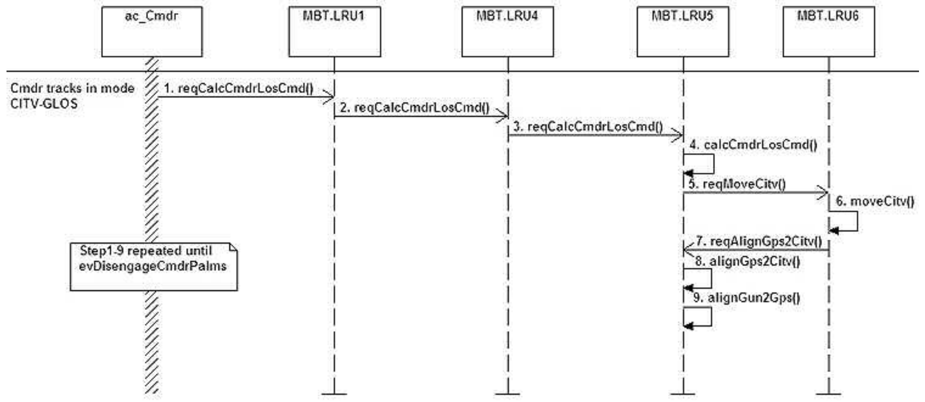
Figure 9B: *Documentation of Physical Subsystem Interfaces by Means of an N^2 Chart*

- Logical interface control document (N^2 chart).
- Subsystem/subsystem component test vectors derived from the system-level use-case scenarios.

Conclusion

For a long time, the UML was considered a modeling language suitable only for software developers that follow the object-oriented paradigm. This article demonstrates that with the release of UML 2.0 – specifically with the introduction of (composite) structure diagrams – the UML could also be applied to function-driven systems engineering. Based on this common, paradigm-independent language for both systems engineers and software engineers, an integrated systems/software development process could be defined, allowing a seamless transition between the two domains. Still, the UML needs some extensions to cover the needs of systems engineers completely (e.g., time-continuous communication). For this purpose, the object management group formed the Systems Modeling Language (SysML) consortium [2]. The first release of the SysML specification will be in the fourth quarter 2005.◆

Figure 10: *White-Box Scenario UC1WB21 (Decomposed Black-Box Scenario of Figure 5)*



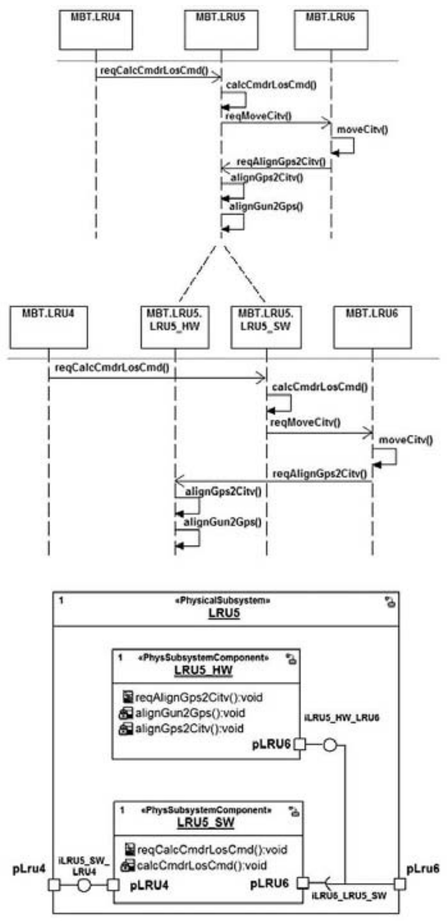


Figure 11: *Subsystem Architectural Design*

References

1. Object Management Group. Unified Modeling Language. "UML 2.0 Specification" 17 Aug. 2005 <www.uml.org/#UML2.0>.
2. SysML Forum. "SysML Specification Vers. 0.9 Draft" 10 Jan. 2005. 17 Aug. 2005 <www.sysml.org/artifacts.htm>.

About the Author

Hans-Peter Hoffmann, Ph.D., is director and chief methodologist for Systems Design at I-Logix Inc., a real-time object-oriented and structured systems design automation tool vendor. Focusing on methodology consulting, Hoffman works as an international consultant for model-based system development. He has 25 years experience in the design and development of complex systems in the aerospace/defense and automotive industries. Hoffmann co-developed the I-Logix Integrated Systems/Software Development Process HARMONY, which combines Unified Modeling Language (UML)/Systems Modeling Language-based systems engineering and UML-based software engineering. Previously as director of the simulation department of the Missile Division at Messerschmitt-Bölkow-Blohm Germany, (now European Aeronautic Defense and Space Company N.V.), he developed a methodology for modeling and analysis of flight control systems.

I-Logix Inc.
3 Riverside DR
Andover, MA 01810
Phone: (978) 645-3022
Fax: (978) 682-5995
E-mail: peterh@ilogix.com

Web Sites

International Association of Software Architects

www.iasarchitects.org

The International Association of Software Architects (IASA) is a nonprofit organization dedicated to the advancement and sharing of issues related to software architecture in the enterprise, product, education, and government sectors. IASA functions as an umbrella organization to chapters spread throughout the world, and as a driving force for research and standards that advance the understanding of software architecture. IASA helps sustain city chapters by forming software architecture user groups and by fostering the sharing of ideas and information between these city groups and industry leaders.

International Enterprise Architecture Center

www.ieac.org

The International Enterprise Architecture Center (IEAC) is a global, independent, nonprofit professional membership organization dedicated to the dissemination of knowledge about enterprise architectures, management, and the marketplace to help members strengthen their performance and better serve society. The IEAC assists in the development of enterprise architecture frameworks, methods and disciplines across all enterprise environments.

Security in a COTS-Based Software System

Arlene F. Minkiewicz
PRICE Systems

Planning and budgeting for the development of a software system composed primarily of commercial off-the-shelf (COTS) components presents unique challenges to those with project and process responsibilities. These challenges are further intensified when security issues are present such as security requirements in the system being developed, security constraints applied to the development team, or both. Because these security issues may impact the functional size of the software being developed, the productivity of the development team, and the ability of the team to communicate, they are important considerations when estimating cost and effort for any software project. They have additional effects when the system is COTS-based. In August's CrossTalk*, work by this author presented a methodology for approaching COTS-based software projects [1]. This article extends that methodology to consider the impacts that security requirements or constraints may impose on each of the activities in the process. It presents an overview of the causes of security vulnerabilities in software and an understanding of how to assess what impact security constraints will have on your COTS-based software projects.*

The developers of software systems are relying on incorporating commercial off-the-shelf (COTS) software components to decrease cost and time to market for delivery of new software systems. Increasingly, the procurers of these software systems are driving this decision to consider COTS solutions for significant functionality. This emergence of COTS solutions as viable and desirable has occurred over time and has not been without trials on both sides. It has taken time for software developers and their customers to begin to understand where COTS solutions will save money and where they will not. Great strides have been made in this area, but there is still much to learn.

As this COTS revolution unfolds, another issue emerges that impacts the entire software industry and has particular ramifications to those developing COTS-based systems. This issue is the increased need for security in software systems driven by rapidly growing networking capabilities introduced by leaps in networking technology and the corresponding leaps in dependence on this technology. According to the National Strategy to Secure Cyberspace:

> Identified computer security vulnerabilities – faults in software and hardware that could permit unauthorized network access or allow an attacker to cause network damage – increased significantly from 2000 to 2002, with the number of vulnerabilities going from 1,090 to 4,129. [2]

The Computer Emergency Response Team Coordination Center at Carnegie Mellon University, which tracks incidents of malicious software intrusions, reports a 2,099 percent increase in incidents from 1998 to 2002. These incidents range from security breaches impacting a single site to those impacting hundreds of sites [3]. These security breaches cost U.S. industry and government billions of dollars.

At the same time that purchasers of software systems are driving solution providers toward possible cost savings with a well-implemented COTS-based solution, they are putting additional demands on the software developers that these solutions meet specific software security criteria. These criteria are generally based on Evaluation Assurance Levels (EALs) as specified in the Common Criteria (CC) for Information Technology Security Evaluation [4] or some analog to these criteria. The CC sets a standard for assigning levels of security compliance of software. This additional requirement for secure software could clearly throw new and unexpected complications into the slowly emerging sense of what drives the costs of COTS-based software projects.

This article addresses the software security issue in general and then in the context of developing COTS-based systems. It begins with an outline of the author's methodology for this research, then describes and bounds the problem being addressed. Next, the issues surrounding software security are addressed. The author then outlines the six steps to a successful COTS implementation and addresses how these steps are impacted by security requirements.

Solution Methodology

The first step in any operations research project is to identify the problem being solved. The problem we are attempting to solve is that of identifying how security constraints impact the activities involved in delivering a COTS-based software system and how these impacts affect project costs. This was accomplished using literature reviews, expert knowledge, and interviews with practitioners to supplement the cost data available.

Once the problem is identified, the next step in constructing a parametric cost-estimating solution is to study and understand the subject process, and from this construct a mathematical model. Research led to the development of a mathematical model based on assumptions of productivity standards for each activity in the process, and adjustments of these standard values to accommodate for the additional rigor and processes associated with elevated security requirements.

This mathematical model was then exercised by software developers and project mangers to determine how it fared when applied to real-life situations. Once satisfied that the model was useful for practitioners, data from various datasets containing both commercial and aerospace data was applied to determine where it worked well and where further work was required.

Bounding the Problem

COTS-based software solutions can be a cost effective method for successfully delivering software systems if these projects are planned with a proper understanding of the activities associated with the implementation and on-going sustenance of COTS-based software systems. These activities and their cost drivers have been well defined in [5]. Adding security constraints to a COTS-based software system will affect the execution of some of these activities and, correspondingly, change how these activity's costs should be evaluated.

In bounding the problem, it is important to understand what definition of COTS is being applied and how security constraints are measured. To have a meaningful discussion on the costs of COTS software, it is important to start with a clear understanding of what is and is not included when we discuss COTS software. For this article, we started with the definition from the University of Southern California's Center for Software Excellence study that led to the Constructive COTS (CoCOTS) model [6]. The definition of a COTS software product follows:

- Commercially available software product – sold, leased, or licensed.
- Source code unavailable but documentation provided.
- Periodic releases with new features, upgrades for technology, etc.

We found this definition too limiting for several reasons. It is not consistent with many of our observations of actual COTS software integration efforts. Customization appears to be quite common, particularly with embedded COTS software. In creating a general-purpose solution, it seemed important that we not overlook the issue of customization. For this reason the definition was altered to include off-the-shelf software with source code available.

We have used the CC as the basis for evaluating security requirements. It was developed to provide a standard for security criteria and evaluation processes of that criteria. The CC contains seven hierarchical sets of assurance requirements called EAL's and named EAL1 through EAL7, with EAL1 representing the least amount of security and EAL7 representing the most security. Increasing degrees of documentation, design rigor, formal processes, etc. are required as you move from EAL1 to EAL7.

Software security issues can be grouped into three categories [7]:

- **Development for Security.** The effect of security constraints on the development of software.
- **Operational Security.** The effect of security policies and processes on the development environment.
- **Physical Security.** The effect of developing software in a secure environment.

The research in this article only covers the first two of these categories. Physical security is left as a future research challenge.

Software Security

In its report issued March 2004, the Software Process Subgroup of the Task Force on Security across the software development life cycle defines the goals of software security:

> The primary goals of software security are the preservation of the confidentiality, integrity, and availability of the information assets and resources that the software creates, stores, processes, or transmits, including the executing programs themselves. [3]

In other words, users of secure software have a reasonable expectation that their data is protected from unauthorized access or modification and that their data and applications remain available and stable. Clearly some applications have a need for a much higher degree of assurance than others. Software designed for standalone desktop use is much less likely to be subject to threats than multi-user software intended for use across a network. Software that is shipping confidential client information or government secrets requires more security than the instant messaging software that is the backbone of teenage culture.

"Despite the advances that the software industry has made in development practices and processes, the buffer overflow continues to be the most frequently cited security vulnerability ..."

As increasing security constraints will increase the cost of developing and maintaining any software system, it becomes an important part of the software acquisition process to understand the required level of assurance based on the functionality the software is intended to perform and the end users who will consume that functionality.

Software security requirements present themselves in two forms. The first are those requirements that impose additional functional requirements for features specifically related to security. Examples of these are encryption algorithms, password protection requirements, or remote access security procedures. The second form of security requirements relate to the additional levels of qualification and testing required to ensure that the software does not allow security breaches into the system on which it operates or the data it maintains. These requirements involve ensuring there are no back doors, buffer overflows, or defects that allow entry to hackers. They also require that patch releases be handled in such a way that the wise hacker cannot use them as road maps to the weaknesses in the software system.

The functional requirements related to security impact cost primarily through increased functional size of the application. When the software is being developed in-house, this additional size drives design, code, and test activities for these requirements. When the software is COTS, this additional size drives integration and test activities.

Before a conversation can take place about the cost impacts of compliance to specific security assurance levels, it is important to understand the various COTS selection strategies that might apply. The selection strategy applied determines where in the COTS implementation process the costs will be incurred and what the extent of those effects might be. There are basically three choices an integrator has when selecting COTS components for a software system:

- Buy and wrap.
- Buy only pre-certified components.
- Buy components and certify internally.

The *buy-and-wrap* option indicates that the integrator intends to purchase the COTS components that best meet the functional requirements for the software. The integrator will then develop a glue code wrapper that operates around this component, isolating it so that any security threats inherent in the COTS component cannot be reached through the resultant applications.

An integrator who opts to *buy only pre-certified components* intends to limit evaluations to COTS components from vendors that have certified their components to the required level of compliance. This reduces the integrator's development and integration effort, but limits the selections and is likely to increase the purchase and maintenance costs of the COTS components. It may also lead to the necessity to develop more in-house functionality if properly certified components are unavailable for some functional requirements.

The option to buy components and

certify internally seems the least desirable of all options. When this path is taken, the integrator purchases the components that best meet their requirements and then has these components certified to the required level of compliance. The integrator can select components that meet functional requirements, but then must bear the burden and expense of the certification process. If the certification fails, the integrator must then restart the evaluation and selection process to identify other potential solutions and must contend with the fact that they have purchased software that has no value.

Security breaches in software are caused by defective specification, design, and implementation [3]. Lack of quality in software creates openings through which threats can be realized. Despite the advances that the software industry has made in development practices and processes, the buffer overflow continues to be the most frequently cited security vulnerability according to a paper published by the Oregon Graduate Institute of Science and Technology [8]. Other defects that lead to security breaches include format bugs, resource leaks, hard-coded path names, and malformed inputs. Additionally, sloppy implementation of encryption algorithms, password transmission routines, and graceful failures lead to security breaches.

Clearly software that has been developed and integrated by organizations that have invested in putting proven software engineering practices and processes in place is going to reach security assurance level compliance with less cost (and less drama) than software developed in a more ad-hoc fashion. The institutionalization of good software development processes is an important factor to consider when planning a project with security constraints.

Another commonly cited cause for vulnerabilities in software is the training and expertise of the developers and integrators of software systems. The successful implementation of secure software systems requires that security and quality be considerations from day one. The skills required to do this are not innate and have not traditionally been stressed as part of a typical computer science or software engineering curriculum. Although this condition is improving with respect to good software development practices, those skills related specifically to security concerns continue to be overlooked in many instances. The cost impact of *fixing* a software system not designed and implemented to be a secure, defect-free system is substantially higher than the cost of developing it from the ground up.

Six Steps to a Successful COTS Implementation

To understand the cost impacts of security constraints on the development of a COTS-based system, it is important to first understand the activities that occur during any successful COTS-based software project. These activities are briefly outlined below. Also, [1] and [4] present more detailed descriptions of this process. Research has indicated the essential activities that must take place to ensure successful COTS-based projects are the following:

- Analyze software requirements.
- Evaluate and select COTS solution(s).
- Negotiate terms with vendors.
- Implement the COTS-based solution.
- Maintain license, subscription, and royalty fees.
- Maintain and upgrade the COTS-based solution.

Analyze Software Requirements

Software requirements analysis should occur regardless of whether the decision is made to build, buy, or borrow. A proper understanding of system requirements that are to be satisfied by software is essential to a successful software project. It is during the software requirements analysis activity that decisions can be made as to which requirements can be satisfied with off-the-shelf components.

Evaluate and Select COTS Solution(s)

Once a decision to pursue a COTS alternative is made, the first step is to determine the availability of COTS solutions that have the potential to provide needed functionality and evaluate these solutions. The evaluation needs to be focused on more than just product characteristics such as functionality, architecture, and technology, but also on vendor characteristics such as maturity, stability, and ability to provide adequate support, training, and documentation.

Negotiate Terms With COTS Vendors

Certainly it is important to negotiate the best deal possible when working with one or more vendors to craft a solution. Vendors are much more likely to address customer concerns with missing or incomplete functionality as well as bugs in the software before they sign on the dotted line.

Implement the COTS-Based Solution

Once an analysis, evaluation, and selection of a COTS-based solution is complete, implementation can commence. Implementation includes the tailoring of the COTS components; modifications to COTS software (if this is possible and desirable); design, code, and test of any glue code required; and higher level integrations of COTS components with other components.

Maintain License, Subscription, and Royalty Fees

License or maintenance fees need to be paid to ensure updates and upgrades as well as continuing support of the COTS components.

Maintain and Upgrade the COTS-Based Solution

Once the software is deployed, there are several ongoing activities required to keep it operational and keep end users happy. These include the ongoing evaluation of upgrades from the vendor, inclusion of those upgrades when desirable, bug fixes in glue code or to compensate for errors in the COTS components the vendor will not fix, and higher level integration as upgrades and bug fixes are deployed.

Security Implications for the Six Steps

The previous section outlined the activities that should take place, and thus should be part of the plan, for the implementation of any COTS-based system. This section looks at the additional factors that must be considered when the COTS implementation includes requirements to comply with security constraints, and highlights how those factors will impact activity costs as compared to the costs of those same activities with nominal security constraints.

Analyze Software Requirements

During the Requirements Analysis activity, functional security requirements need to be analyzed, along with the non-functional ones. From a cost and planning perspective, these additional requirements should be modeled as an additional functional size that needs to be either developed or purchased and integrated. Functional size can be any measure of software size that relates to the amount of user functionality that is being delivered such as function points or feature points. An additional cost consideration is the amount of skill and expertise the integration team has with implementing secure software systems. It is during the requirements analysis phase that the COTS selection strategy is likely to be determined.

Evaluate and Select COTS Solution(s)

The COTS selection strategy plays a big part in determining the impact of security constraints on this activity. If the strategy is to buy and wrap COTS components, then the evaluation and selection activity effort should focus entirely on the functional requirements for which COTS components are being considered, with little attention paid to the security requirements since this will be taken care of externally to the COTS components. The cost impacts of this activity would be a function of the total number of COTS solutions available for consideration, along with the number expected to be selected for more detailed consideration. The decision to build a wrapper also impacts the cost of glue code development (described below) as the need for wrapper code increases the amount and complexity of glue code that must be developed.

If the strategy is to buy pre-certified components, you would expect this to have substantial impact on the amount of time and effort associated with this activity. This strategy is likely to result in a broader initial search for qualified COTS components that meet the desired functionality and security constraints, but then a smaller set of candidate components available for detailed evaluation. The cost and effort impacts to this activity will be a function of the security assurance level required and the number of COTS components available for evaluation that are certified to that level.

The strategy to buy and certify components is a bit riskier to predict with respect to effort or cost because there is an inherent risk that the evaluation process may be revisited if internal certification fails for selected components. Integrators should take steps to mitigate this risk by only evaluating components from vendors with documented software development practices and proven quality records. With such risk mitigation strategies in place, this activity would consist of a broader initial search, as all available components are contenders followed by a more detailed evaluation of the few that appear to meet high quality standards. Additionally, certification comes with its own costs, including fees for laboratory testing, modifications identified as the result of this testing (if modifications are possible and desired), collecting documentation required for assurance, and fees to certification agencies.

The cost and effort impacts to this activity will be driven by the security assurance level required, the number of COTS components likely to be certified to that level, the functional size and complexity of the COTS components that require certification, and the state and availability of documentation.

Negotiate Terms With COTS Vendors

If the strategy is buy and wrap, then negotiations with vendors should not be impacted by security constraints. Vendor pre-certification is unlikely to impact the negotiations process either, unless the pre-certification is being done as part of the negotiated terms of the contract. If the certification is being performed by the integrating organization, the terms of the contract should include actions to be taken if COTS components fail to meet any promises made with respect to quality or security.

"It is during the software requirements analysis activity that decisions can be made as to which requirements can be satisfied with off-the-shelf components."

Implement the COTS-Based Solution

The implementation activities for a COTS-based solution can be affected by security constraints in various ways. As with previous activities, the selection strategy determines which activities are affected and to what extent.

- **Tailoring of COTS solution.** Security-specific tailoring tasks should be handled as part of the functional requirements for the system. The impact to the cost of this activity would be a function of the functional size imposed through those requirements.
- **Modification of COTS software.** If the buy and wrap strategy is in place, the costs of modifications should not change regardless of security requirements as the wrapper has isolated the COTS component from the rest of the system in a secure cocoon. If the COTS components are pre-certified or internally certified and then modified by the integrator, an entire recertification must be accomplished, most likely at the cost of the integrator. The cost and effort for this activity would be driven by the security assurance level required and the functional size of the COTS component(s) that are modified. Institutionalized software development practices and security expertise would mitigate some of the costs of this process.
- **Design, code, and test of glue code.** When a buy-and-wrap strategy is employed, the glue code size should be expanded to include the amount of wrapper code that must be developed. The cost of the development of this wrapper code, along with any other glue code that is written, will be impacted by the security assurance level required. This impact can be substantially mitigated when software development practices are institutionalized and when the integration team has training and experience in developing secure software.
- **Integration of COTS components with other COTS or custom components.** Integration and test are where the rubber meets the road. As with other non-functional requirements, security requirements are not fully testable until all the pieces of the system are working together. It is possible that certain combinations of components create security vulnerabilities where none existed within the individual components. Regardless of selection strategy, this activity will be impacted. Extent of this impact is determined by the security assurance level required, but is mitigated when good software development practices are institutionalized and the integration team contains members with security training and experience.

Maintain License, Subscription, and Royalty Fees

If the strategy is buy and wrap, this activity is unlikely to be impacted by security assurance requirements. If COTS components are pre-certified or internally certified, renewal time is the time to ensure that promises with respect to security and quality are being maintained as the COTS products are upgraded over time.

Maintenance and Upgrade of the COTS-Based Solution

Maintenance of any COTS-based system can be problematic, particularly as the number of different vendors increases. Security constraints further complicate the maintenance and upgrade activities.

- **Evaluation and inclusion of updates and upgrades from the vendor.** When a buy-and-wrap strategy is employed, the inclusion of upgrades may require modifications to wrapper code to accommodate new or changed interfaces. Upgrades for internally certified components would require recertification. One would expect that precertified components would recertify with each upgrade and update. Regardless of selection strategy, upgrades and updates require some level of integration and test at the system level to ensure that security constraints are still met by the system. The cost of this activity is driven by the security assurance level, but is mitigated by good development practices, training, and expertise.
- **Bug Fixes.** Regardless of selection strategy, bug fixes will require some level of reintegration and test, which requires reverification at the system (or subsystem) level that these fixes have not introduced security vulnerabilities. Additionally, if fixes are made to COTS components or to modifications made to those components, the entire certification process for that component must be recertified. The cost and effort of this activity is driven by the security assurance level, as well as the amount of glue code and modified code being maintained. This cost can be mitigated with the institutionalization of good software development practices and an integration team with security training and expertise.

Conclusions

More and more, integrators are being asked to deliver software systems that meet high-level security constraints while delivering most (if not all) of their functionality using off-the-shelf components. As integrators are just beginning to understand all of the issues associated with a successful COTS-based software project, they now have to understand how security assurance requirements will impact those issues and what new issues will be introduced.

Security constraints impact a project in two ways. Functional security requirements increase the functional size of the software system being developed and need to be treated in the same way as all other functional requirements being met by COTS components or homegrown code. Non-functional security requirements to attain a specific level of security assurance require additional processes, documentation, testing, and verifications. The additional focus on these activities will result in additional cost to COTS integrators. Most security vulnerability in software is a result of poor quality and poor or inconsistent software development practices. By improving these practices and employing people with expertise in security and excellent software development skills, integrators can reduce the cost increases driven by requirements for security.

This article reviewed the impacts of security constraints on the cost of a project to deliver COTS-based software systems. It reviewed these impacts in the context of the activities required to successfully implement a COTS-based system and highlighted those areas where security constraints apply. The security-related factors found most likely to impact costs were the COTS selection strategy employed and the security assurance level required. In organizations where investments have already been made to institutionalize good software development processes and hire people with the training and experience to deliver secure, defect-free software, the cost impact of security requirements is less severe.◆

References

1. Minkiewicz, A. "The Six Steps to a Successful COTS Implementation." CrossTalk Aug. 2005 <www.stsc.hill.af.mil/crosstalk/2005/08/0508Minkiewicz.html>.
2. The White House. "National Strategy to Secure Cyberspace." Feb. 2003 <www.whitehouse.gov/pcipb>.
3. Software Process Subgroup of the Task Force on Security Across the Software Development Life Cycle. "Process to Produce Secure Software." National Cyber Security Summit, Mar. 2004.
4. The National Institute of Standards and Technology and the National Security Agency. "Common Criteria for Information Technology Security Evaluation, Vers. 2.1." National Institute of Standards and Technology. Aug. 1999 <http://csrc.nist.gov/cc>.
5. Minkiewicz, A. "The Real Costs of Developing a COTS-Based System." Proc. for IEEE Conference on Aerospace and Defense, Big Sky Montana, Mar. 2004.
6. Center for Software Engineering. "CoCOTS White Paper." Los Angeles, CA: University of Southern California, June 1997 <http://sunset.usc.edu/research/COCOTS/cocots_main.html>.
7. Reifer, D., et al. "Estimating the Cost of Security for COTS Software." 2nd International Conference on COTS-Based Software Systems. Los Angeles, CA. Mar. 2003.
8. Festa, P. "Study Says Buffer Overflow Is Most Common Security Bug." C-Net news.com 23 Nov. 1999 <http://news.com.com>.

About the Author

Arlene F. Minkiewicz is chief scientist of the Cost Research Department at PRICE Systems. She is responsible for the research and analysis necessary to keep the suite of PRICE estimating products responsive to current cost trends. In her 20-year tenure with PRICE, Minkiewicz has researched and developed the software cost estimating relationships that were the cornerstone for PRICE's commercial software cost estimating model, ForeSight, and invented the Cost Estimating Wizards originally used in ForeSight that walk the user through a series of high-level questions to produce a quick cost analysis. As part of this effort, she invented a sizing measurement paradigm for object-oriented analysis and design that allows estimators a more efficient and effective way to estimate software size. She recently received awards from the International Society of Parametric Analysts and the Society of Cost Estimating and Analysis. Minkiewicz contributed to a new parametric cost estimating book with the Consortium for Advanced Manufacturing International called "The Closed Loop: Implementing Activity-Based Planning and Budgeting," and she frequently publishes articles on software estimation and measurement. She has been a contributing author for several books on software measurement and speaks frequently on this topic at numerous conferences.

PRICE Systems
17000 Commerce PKWY STE A
Mt. Laurel, NJ 08054
Phone: (856) 608-7222
Fax: (856) 608-7247
E-mail: arlene.minkiewicz@pricesystems.com

Software Component Interoperability

Jeffrey Voas
SAIC

When a software system fails, a confusing and complex liability problem ensues for all parties that have contributed software functionality (whether commercial off-the-shelf [COTS] or custom) to the system. This article explores the interoperability problems created by defective COTS software components, and, in particular, the hidden interfaces and non-functional behaviors. It also looks into the problem of composing non-functional behaviors that are related to quality-of-service attributes.

When there are complaints about the quality of the software, they are usually directed at the software's behavior, and not the underlying text [1]. Therefore it is important to define what constitutes bad behavior in software.

Bad software behavior is a function of the *environment* that the software resides in, and the environment is a mixture of the underlying hardware, the operating system, potential threats, the operational profile, other external software components that interact with the software, and the manner in which the software will be used [2]. Since these factors affect the software system's behavior, the architectural plan for how components are to be integrated should be, in part, based on these factors.

For example, a toaster comes with a warranty and that warranty assumes a particular environment, such as sitting on a kitchen countertop. Now take a working toaster, put it in another environment, like a bathtub full of water, and the behavior of the toaster is not going to be the same, and the warranty no longer applies. The same argument applies to software.

Will the Real Operational Profile Please Stand Up?

One of the most important issues during requirements generation is defining, as best as possible, the software environment; its operational profile is key.

Each piece of software has a set of input vectors that may be executed during testing or field usage. That is the software's *input domain*. The *operational profile* is simply a probability distribution function (PDF) for the input vectors of the input domain. This means each input vector has a certain probability of being chosen during testing or field usage, and the PDF is what defines the probabilities for each input vector. For example, if the input domain has 10 input vectors, and each input vector is as equally likely to be selected as any other vector, then each vector has a 10 percent chance of being selected. (Note that the operational profile is highly important when system-level reliability testing occurs [3].)

Thus, without an accurate description of the operational profile, predictions concerning how the software will behave in the field are unlikely to be accurate since each individual input vector can cause differing behaviors. Therefore, spending the extra time to contemplate the eventual operational profile and target environment is often as important as defining the software's functionality during requirements definition.

Tolerating Component and Subsystem Failures

More and more software is delivered to system integrators in black-box form; these components are packaged as executable objects (with licensing agreements that forbid decompilation back to source code), e.g., dynamic link libraries that originate from third-party sources, e.g., commercial off-the-shelf (COTS) software. A worthy goal, then, is to provide a methodology for determining how well a system can perform when particular black boxes are of such poor quality that interoperability and integration problems are almost inevitable.

One technique for assessing the level of interoperability between COTS software components and custom components is called Interface Propagation Analysis (IPA) [4]. IPA perturbs (i.e., corrupts) the states that propagate through the interfaces that connect COTS software components to other components. IPA is one form of *software fault injection* [5]. By corrupting data going from one component to a successor component, failure of the predecessor is simulated, and its impact on the successor can be assessed. This approach allows for measuring the level of intolerance when one component fails, sending junk information (or even a lack of information) to its neighbor.

To modify the information (states) that components use for inter-component communication, write access to those states is required (to modify the data in those states during the simulation). This is done by creating a small software routine called PERTURB that replaces the original output state with a different (corrupted) state. This is, of course, done as the system executes. By simulating the failure of various software components, we can assess whether the remainder of the system can tolerate it.

I will illustrate this by using Advanced IBM Unix's (AIX) cos() function in this example. Note that AIX's cos() is a fine-grained COTS utility for which we do not have access to the source code:

```
double cos(double x)
```

This declaration indicates that the **cos()** function receives a double integer (contained in variable x) and returns a double integer. Because of C's language constraints, the only output from **cos()** is the returned value, and hence that is all that IPA fault injector can corrupt.

To see how this analysis works, consider an application that contains the following code:

```
 if (cos(a) > THRESHOLD)
{
 do something
   }
```

The goal, then, is to determine how the application will behave if **cos()** returns incorrect information. To do so, the return value from the call is modified:

```
 if (PERTURB(cos(a)) > THRESHOLD)
{
 do something
    }
```

Note that IPA is more than just an

interesting research idea. It has been used successfully in a variety of critical software systems that required better techniques to perform impact analysis and assess potential interoperability conflicts; those case studies are compiled in [5].

Composing *ilities*

Clearly, today's COTS components are much more substantial in functionality and complexity than the previous AIX example. One of the real problems in composing today's COTS components that have advanced functionality and complexity is combining large components that each have varying quality-of-service (QoS) attributes into one component that now inherits a new set of QoS attributes.

Much of the work from the past 10 years into component-based software engineering (CBSE) and component-based development (CBD) has dealt with functional composability (FC). FC is concerned with whether the following is true:

F(A) _ F(B) = F(A _ B)

where,

_ is some mathematical operator

That is to determine whether a composite system, F(A _ B), is created that has the desired functionality after joining A and B. Therefore, A and B now have a way to communicate, whether one-directional or bidirectional. Instead of acting alone, they now act together as one unit.

But, increasingly, the software community is discovering that FC, even if it were a solved problem, is not mature enough for other serious concerns that arise in CBSE and CBD such as the problem of composing *ilities*. *Ilities* are non-functional, QoS properties of software components and they define characteristics such as security, reliability, fault-tolerance, performance, availability, safety, testability, survivability, maintainability, etc. The properties, as well as others, are what ultimately determine whether the software is well-behaved or not.

The problem stems from our inability to know a *priori*, for example, what the security of a system composed of two components – A and B – will be even if we have knowledge about the security capabilities built into A and knowledge about the security capabilities of B. Why? Because the security of the composite system is based on more than just the security of the individual components.

For example, suppose that A is an operating system and B is an intrusion detection system. Operating systems usually have some level of authentication security built into them, and intrusion detection systems have definitions for the types of event patterns that likely warn of an attack. Thus the security of the composition of these two components depends on the security models built into the individual components.

But even if A has a worthless security policy or flawed implementation, the composite can still be secure. How? By simply making the performance of A so poor that no one can gain access, i.e., if the intrusion detection system is so inefficient at performing an authentication, then in a strange way, security is actually offered. And if the implementation of A's security mechanism is so unreliable that it disallows access to all users, even legitimate ones, then strangely security is again increased. While these last two examples are not a reasonable way to achieve higher levels of system security, both do actually decrease the likelihood that a system will be successfully attacked.

Using the same example of A and B, this time assume that A provides excellent security and B provides excellent security. The usefulness of B's security mechanisms is a function of calendar time because new threats and ways to intrude are always being discovered. So even if you could create a scheme that determines the following:

Security(A) _ Security(B) = Security (A _ B)

the security offered by B is always a function of which version of B is being composed with A, what recent new threats have arisen, and how many of these new attack patterns have been built into B.

So the question then comes down to which ilities, if any, are easy to compose? The answer is that there are no ilities that are easy to compose, and some are much harder than others. Further, there are no widely accepted algorithms for how to do so.

For this problem applied to reliability, consider a 2-component system in which component A feeds information in B, and B produces the output of the composite (two components in series). Assume that both components are reliable. What can we assume about the composite's reliability? While this information certainly suggests that the composite system will be reliable, it must be recognized that components (which were tested in isolation for their individual reliabilities) can suddenly behave unreliably when connected to each other, particularly if the isolated test distributions did not at all reflect the type of information that A will be sending to B.

This brings us back to the importance of understanding the environment and operational profile for each component. Further, there can be non-functional behaviors that cannot be observed nor manifest themselves until after composition occurs. Such behaviors can undermine the reliability of the composition. Finally, if one of the components is simply the wrong component – although highly reliable – then naturally the resulting system will be useless.

In addition to reliability and security, one *ility* that, at least on the surface, appears to have the best possibility of successful composability is performance. But even that is problematic from a practical sense. The reason stems from the fact that even if a performance analysis was performed on a single component, the practical consequences on that component's performance after a composition with another component may depend heavily on the hardware and other physical resources. That could require that different hardware variables might need to be dragged along with a certificate that states only minimal, worst-case claims about the performance of the component.

For example, in the pharmaceutical industry, drugs come with endless warnings/rules as to who should take them and who should not. In the discussion here, the hardware variables are the various other medical circumstances of an individual slated for a particular drug (liver problems, heart problems, etc.). Clearly, this complex issue creates serious pragmatic difficulties and again takes us back to the need for a precise definition of what is the real environment of a software component [2, 6]. Today, the definition of what is the real environment of a component is an open question, needing new thinking and new research efforts [7].

Note that non-functional behaviors are particularly worrisome in COTS software products. Non-functional behaviors can include malicious code (Trojan horses, logic bombs, etc.) and any other behavior or side effect that is not documented.

Another worrisome problem facing CBSE and CBD is hidden interfaces. Hidden interfaces typically are channels through which application or component software is able to convince the operating system to execute undesirable tasks or processes.

Interestingly, IPA can partially address the issue of detecting hidden interfaces and non-functional behaviors by forcing software systems to reveal those behaviors after the input stream of a COTS component receives corrupted input. Injecting corrupt information into a component can possibly force it to execute different (and rarely executed) code paths, flushing out behaviors that would not normally occur with uncorrupted information.

The reason that IPA can only be partially successful in doing so is that IPA is a function of (1) the number of corrupted inputs used, and (2) the number of executions of the software. Just as *exhaustive testing* is the only way to guarantee correctness via testing, IPA is limited in the amount of analysis it can perform to detect hidden interfaces and non-functional behaviors. In short, you cannot test every potential combination of ways that you can inject corrupted data to see what results, and where those results propagate.◆

References

1. Voas, J., and C. Vossler. "Defective Software: An Overview of Legal Remedies and Technical Measures Available to Consumers." Academic Press 53 (2001): 451-497.
2. U.S. Patent No. 6,862,696: System and Method for Software Certification. 1 Mar. 2005.
3. Voas, J. "Would the Real Operational Profile Please Stand Up?" IEEE Software 17.2 (Mar. 2000): 87-89.
4. Friedman, M., and J. Voas. Software Assessment: Reliability, Safety, Testability. New York: John Wiley & Sons, 1995.
5. Voas, J., and G. McGraw. Software Fault Injection: Inoculating Programs Against Errors. New York: John Wiley & Sons, 1998.
6. Voas, J. "Certifying Off-the-Shelf Software Components." IEEE Computer 31.6 (June 1998): 53-59.
7. Whittaker, J., and J. Voas. "Towards a More Reliable Theory of Software Reliability." IEEE Computer 33.12 (Dec. 2000): 36-42.

About the Author

Jeffrey Voas is director of Systems Assurance Technologies at Science Applications International Corporation (SAIC). Before joining SAIC, he was the chief scientist at Cigital. Voas has been highly active in the software engineering research community for over 15 years. He has served on numerous journal and magazine editorial boards, written more than 125 papers, co-authored two books, and is currently the 2005 Institute of Electrical and Electronics Engineers Reliability Society president.

SAIC
Crystal Gateway #4
200 12th ST S STE 1500
Arlington, VA 22202
Phone: (703) 414-3842
Fax: (703) 414-8250
E-mail: jeffrey.m.voas@saic.com

The Joint Services
STC
Systems & Software Technology Conference
1 - 4 May 2006 • Salt Lake City, UT

Transforming: Business, Security, Warfighting

Call for Exhibitors

A Few of the Many Reasons to Exhibit at SSTC
- Access to over 2,500 leaders and decision makers in the DoD, Government, Industry, and Academia
- 95% of attendees visit the trade show
- 72% of attendees plan to pursue business with vendors they meet at SSTC
- 88% of vendors say SSTC helps them generate quality leads
- 71% of attendees say they come to SSTC to investigate new strategies and technologies

Similar to the theme for this year – SSTC is transforming!
Have a SPIN, Workshop, or "Birds of a Feather" meeting you'd like to simplify yet magnify?

Ask us how your organization's agenda can be part of SSTC by emailing stc-info@ext.usu.edu or calling 435-797-0047

Register to be an Exhibitor today!

The Eighteenth Annual Joint Services Systems & Software Technology Conference is co-sponsored by:

United States Army
United States Marine Corps
Department of Navy
United States Air Force
Defense Information Systems Agency

Full Conference Schedule will be Posted mid-December at
www.sstc-online.org
800-538-2663

Design? We Don't Need No Stinkin' Design! (or "How to Fail Without Really Trying")

Let me start by pointing out that I am a compulsive list-maker and planner. As a military brat, I grew up overseas. Recently, I decided to take a vacation to Istanbul, Turkey where my father had been stationed from 1964 to 1966. To me, a large part of the fun of a vacation is the preparation and planning. Making lists – what to carry, what sites to visit ranked by importance (with categories of Essential, Important, or Optional) – is exhilarating. To me, half of the fun is examining all the tourist guides, and savoring the experience of the trip by preparing for it.

When I showed my co-workers my list of sites I wanted to visit, sorted both by district and importance, the word *geek* kept creeping up in their responses.

On the other hand, these obsessive-compulsive traits seem to be a good thing when talking about design.

When you mention *design* to most developers, they start thinking of program design. Now program design is indeed one part of design, and a very important part. Program (or module) design is where you convert the algorithms and logic of the design into executable code, assuming that the algorithm and logic were available – assuming that the developer doesn't know more than the designer – and *improve* the logic.

Module design is the easy part of design, however. Design, depending upon the source and reference cited, consists of at least four phases: architecture design, interface design, data design, and module design. This ordering also captures the correct order of importance and difficulty.

Architectural design is where a *grand designer* sets forth the vision of the system. It includes what the major subsections are, the major functionality, and general *feeling* of how the overall system is going to work. Even with automated tools, this step is hard. It is typically said that the architectural design should have a feel to it that says, "Well, of *course* this is the way that the system is supposed to look." It should appear simple. Unfortunately, as system designers know, simplicity is very difficult to achieve. Without a good architecture, however, the system never seems to work well – tasks have to be shared across major subsystems, increasing complexity, and decreasing cohesion.

As part of the architectural design, interfaces have to be established and enforced. In a perfect world, this step should be relatively simple. Unless, of course, you are dealing with legacy systems that already have existing interfaces; interfaces that were, at best, designed using 30-year-old functional decomposition, and do not fit well with object-oriented languages of the present.

When I teach design, I tell my students that 75 percent of all errors will eventually be traced to poor interfaces. Most don't believe me at first, but eventually find out that I was right. The problem with interface incompatibilities is that they often don't show up during unit testing, and cannot be discovered until integration testing. Of course, the later in the life cycle that errors are discovered, the more expensive they are to fix.

Do the words "Boyce-Codd Normal Form" mean anything to you? How about "First, Second, or Third Normal Form"? If so, you're in the minority. These terms refer to relationships between data and keys. Twenty years ago, you wouldn't have dreamt of designing a database without having an expert determine an optimum arrangement of keys and relationships. Now, unfortunately, cheap disk storage and fast processors allow data to be *thrown together*. However, as applications evolve, grow, and are modified due to changing and new requirements, bad data design can slow down your application. It can also contribute to problems with duplication of data, along with consistency and correctness of the data.

In short: design is hard, very hard. The larger and more complex the application, the more important design becomes. If you have a small application that's going to exist on the Web for a few months at the most, design is no problem. On the other hand, if you have a $10 million project, projected to interface with existing Department of Defense systems with an expected life of 20 years, design is critical.

Of course, you can probably get a working system without much design. Enough to get user sign-off. Of course, the first time you try and make any modifications or additions to the system, the lack of a design will cause it to fall apart like a house of cards.

Not to worry. There's always time and money to do it over with a good design, right?

— David A. Cook, Ph.D.
Senior Research Scientist
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Total System Integration...

..SMXG understands the importance -- as leaders in the avionics software industry, SMXG has a proven track record of producing software On-Time, On-Budget, and Defect Free. Our staff of engineering professionals, as well as our industry partners, are committed to providing our customers with software and engineering solutions that make our Warfighters the most dominant forces in the world.

Weapon/Systems Integration

The expertise, software, weapons, interfaces, and aircraft systems that are fully integrated to ensure dependable war-winning capability.

Avionics Integrated Support Facilities
- B-1B, B-2, B-52, E-3, Missiles

Operational Flight Programs (OFP)
- B-1B, B-2, B-52, E-3, KC-135,
- ACM, ALCM, CALCM

Test Program Sets (TPS)
- B-1B, B-2, B-52, E-3, C-5, C-17,
- C-135, C-141, F-15, F-16, F-117,
- KC-135

Areas of Expertise:

- *Navigation*
- *Radar*
- *Control and Display*
- *Simulations and Prototyping*
- *Mission Planning*
- *Defense Management System*
- *Weapons and System Integration*
- *Electronic Warfare*
- *Operational Flight Software*
- *Weapon Delivery and Integration*
- *Systems Engineering*
- *Network Centric development*
- *Turn-Key Test Systems and Solutions*

- *TEST PROGRAM SET Development and Sustainment*
- *INTERFACE TEST ADAPTER (ITA) Design and Manufacturing*
- *AUTOMATED JET ENGINE TESTING (Hardware and Software Development)*
- *Engine Trending and Diagnostics*
- *Software Control Center (SCC)*

Oklahoma City Air Logistics Center
76th Software Maintenance Group
Kevin D. Stamey (Director)
(405) 736-4618
DSN 336-4618
kevin.stamey@tinker.af.mil
www.BringItToTinker.com

CrossTalk is co-sponsored by the following organizations:

Homeland Security

CrossTalk / 309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737